

C++ ve Nesneye Dayalı Programlama

Binnur KURT
kurt@cs.itu.edu.tr



BİLİŞİM ENSTİTÜSÜ
Bilişim Teknolojileri
Tezsiz Yüksek Lisans Programı

C++ ve NESNEYE DAYALI PROGRAMLAMA

1



PROGRAM

1. Giriş

Nesneye dayalı programlamanın (OOP) temel felsefesi,
Yordamsal (procedural) programlama yöntemi ile karşılaştırma,
OOP'den beklenen yararlar ve karşılaşılabilen problemler.

2. C++ dilinin standart C'ye getirdiği yenilikler

C++'ın nesneye dayalı olmayan yeniliklerinin tanıtılması:

- Erim operatörü (::),
- Blok içinde yerel değişken tanımlayabilme,
- **inline** fonksiyonlar,

C++ ve NESNEYE DAYALI PROGRAMLAMA

2

- Fonksiyon parametrelerine başlangıç değeri atayabilme,
- Referans operatörü (&),
- Dinamik Bellek kullanımı : **new** ve **delete** operatörleri,
- Fonksiyonlara işlev yükleme (= Function overloading),
- Operatörlere işlev yükleme (= Operator overloading),
- Fonksiyon şablonları (= Function Templates).

3. Sınıf (= Class) yapısı

- Sınıfların ve nesnelerin (= Object) oluşturulması,
- Üyelere (= Members) erişimin denetlenmesi (= Access control),
- Standart Kurucu fonksiyonlar (= Constructor),
- Özel amaçlı kurucu fonksiyonlar,
- Nesne dizileri,
- Yokedici fonksiyonlar (= Destructor),

- İç içe nesne yapıları (= Nested objects),
- Sabit (= Constant) ve statik üyeler,
- Operatörlere yeni işlevlerin yüklenmesi (= Operator overloading),
- Özet: Sınıf yapısından beklenen yararlar.

4. Kalıtım (= Inheritance)

- Kalıtımın kullanım amacı :
tekrar kullanılabilirlik (= Reusability) kavramı,
- C++'da kalıtım mekanizmasının oluşturulması,
- Kalıtım ile oluşturulan üyelere erişimin denetlenmesi,
- Kalıtımın kurucu ve yok ediciler üzerindeki etkisi,
- Çoklu kalıtım (= Multiple inheritance)
- Nesne işaretçileri (= Pointers to objects).

5. Çok Şekillilik (= Polymorphism)

- Çok şekillilik kavramı,
- Sanal (= Virtual) fonksiyonlar,
- Çok şekillik mekanizmasının C++'da oluşturulması,
- Aynı programın çok şekillik kavramı kullanılmadan ve kullanılarak yazılması,
- Sanal Kurucu Fonksiyonlar.

6. Parametrik Çok Şekillilik (= TEMPLATES)

7. C++ standart giriş/çıkış sınıfları ve nesneleri : Streams

- Tuştakımı/ekran işlemleri,
- Disk (dosya) yazma/okuma işlemleri.

8. Ayrıcalıklı Durum Yönetimi (= Exception Handling)

YAZILIM MÜHENDİSLİĞİ

Yazılım Nedir ?



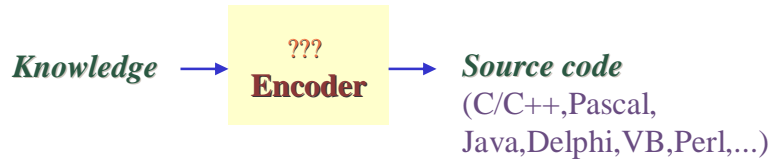
Carnegie Mellon
Software Engineering Institute

<http://www.sei.cmu.edu.tr>

Software \equiv Knowledge Coding

"The Business of Software" – Phillip G. Armour

Comm. Of the ACM, pp. 13, Vol.44, No.3, March 2001



örnek:

$$ax^2 + bx + c = 0$$

knowledge

$$x_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$
$$x_2 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Encoder

```
cout << endl << "Enter a" ;  
cin >> a ;  
cout << endl << "Enter b" ;  
cin >> b ;  
cout << endl << "Enter c" ;  
cin >> c ;  
D = sqrt( b * b - 4 * a * c ) ;  
x1 = ( -b - D ) / ( 2 * a ) ;  
x2 = ( -b + D ) / ( 2 * a ) ;  
cout << "Roots are " ;  
cout << x1 << " , " << x2 ;
```

YAZILIM

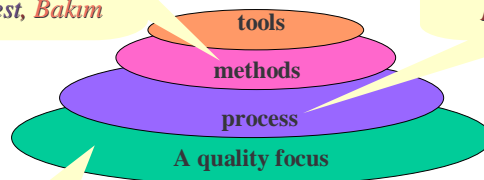
a → x_1
b → x_2
c →

YAZILIM MÜHENDİSLİĞİ

SE is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

*İstemlerin Analizi, Sistem
Analizi, Tasarım,
Kodlama, Test, Bakım*

*Process
Management*



*Software Quality
Metrics*

Layered Technology

www.ygm.itu.edu.tr

SÜREÇ YÖNETİMİ



Carnegie Mellon
Software Engineering Institute

<http://www.sei.cmu.edu.tr>

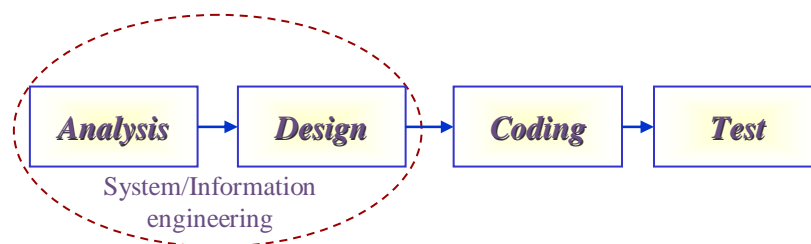
Process Maturity-CMM

Software Engineering Institute's Capability Maturity Model

- Level 1 : Initial
- Level 2: Repeatable
- Level 3: Defined
- Level 4: Managed
- Level 5: Optimizing

SÜREÇ YÖNETİM MODELLERİ

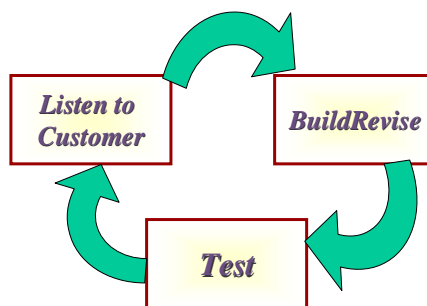
The Linear Sequential Model



YAZILIM YAŞAM ÇEVİRİMİ

System/Information Engineering and Modeling
Software Requirement Analysis
Design
Code Generation
Testing
Maintenance

PROTOTYPING MODEL



NESNEYE DAYALI X

X =

- Veri Tabanı
- İşletim Sistemi
- Kullanıcı Arayüzü
- Grafik Sistemleri
- Benzetim Sistemleri
- Video Kodlama

·
·
·

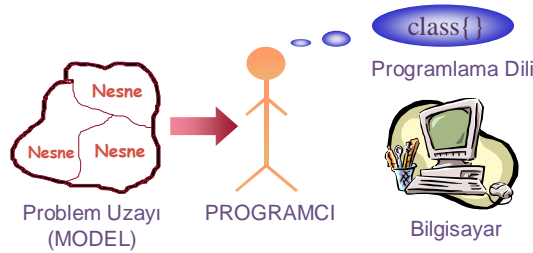


C++ ve NESNEYE DAYALI PROGRAMLAMA

13

NESNEYE DAYALI PROGRAMLAMA

Program geliştirme, problemi oluşturan bileşenlerin modellenmesini ve problemin bu model kullanılarak bilgisayar üzerinde çözüm süreçlerini kapsar. Programlar ise tanımlanan modelin bilgisayarda gerçekleştirme metodlarını tanımlar.



Problemler genellikle (fiziksel/soyut) nesneler içermektedir. Bu nedenle problemlerin nesnelerle modellenmesi, çözümün daha kolay ifade edilmesini sağlamaktadır.

C++ ve NESNEYE DAYALI PROGRAMLAMA

14

Yordamsal Diller – Nesneye Dayalı Diller

- Pascal, C, BASIC, Fortran, ve benzer geleneksel diller yordamsal (= Procedural) dillerdir : Her bir satır bilgisayara ne yapması gerektiğini söyler.
- Program fonksiyonlara bölünmüştür ve her bir fonksiyon iyi tanımlanmış bir işleve ve programdaki diğer fonksiyonlarla iyi tanımlanmış bir arayüze sahiptir.
- Yordamsal dillerde fonksiyonların gerçekleşmesine önem verilir. Verinin organizasyonu ikinci sırada gelir. Ancak verinin varlığı programın varlığının nedenidir.
- Bu nedenle yapısal dillerin bazı önemli sorunları vardır.

Yapısal Dillerin Sorunları

- Global değişkenlere, üzerinde işlem yapmaması gereken fonksiyonlar tarafından erişilebilir.
- Değişkenlerin koruması yoktur. Değişkene, sadece belirli bazı fonksiyonlar tarafından erişimine olanak sağlayacak bir mekanizma yoktur.
- Yeni bir veri yada veri yapısı eklendiğinde, bu veri üzerinde işlem yapan tüm fonksiyonlar yeniden, eklenen veriyi işleyecek şekilde güncellenmelidir.
- Yeni veri tipleri oluşturmak zordur.
- Problemleri güçlü bir şekilde modelleyemezler.

Nesneye Dayalı Yaklaşım

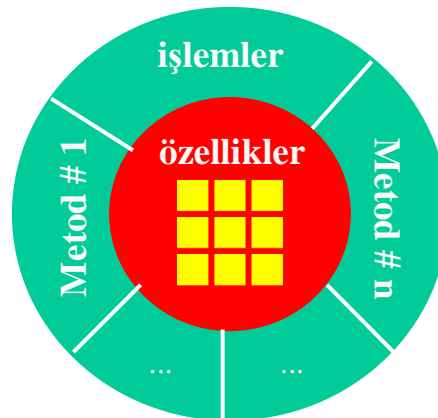
- Veri birincil derecede öneme sahiptir.
- Amaç veri ile bu veri üzerinde işlem yapan fonksiyonları bir araya getirerek yeni bir yapı oluşturmaktır : **SINIF**.
- Veriye erişmenin tek yolu **SINIF** fonksiyonlarını kullanmaktır. Bunun dışında veriye erişim engellenmiştir. Bu şekilde veri gizlenerek yanlışlıkla değiştirilmesinin önüne geçilmiş olur.
- Veri ve bu veri üzerinde işlem yapan fonksiyonlar paketlenmiştir. **Paketleme** (*encapsulation*) ve **veri gizleme** (*data hiding*) nesneye dayalı programlamanın önemli iki kavramını oluşturmaktadır.
- Veri işlenmek istenildiğinde, bu veri üzerinde hangi üye fonksiyonların işlem yaptığı tam ve kesin olarak bilindiği için fonksiyonların gerçekleşmesi, hata ayıklama, kodun güncellenmesi gibi Yazılım Yaşam Çevrimi adımları kolaylaşmaktadır.
- Tipik olarak bir C++ programı birbirleri ile üye fonksiyonları aracılığı ile etkileşen nesnelerden oluşur.

Paketleme ve Veri Gizleme

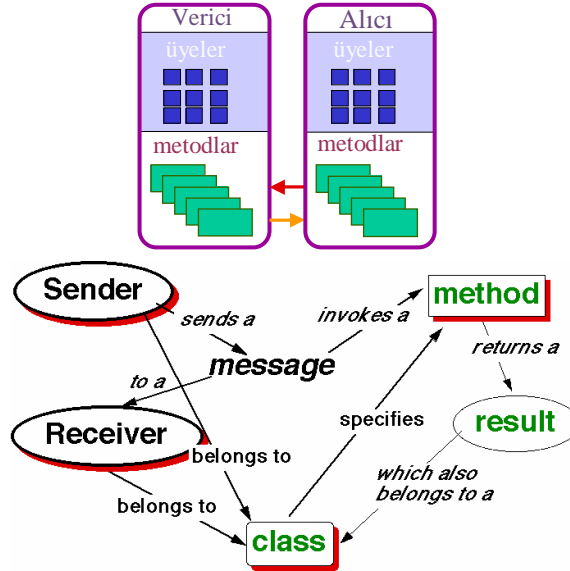
Paketleme



Veri gizleme



Nesneler Arası İletişim



C++ ve NESNEYE DAYALI PROGRAMLAMA

19

Nesneye Dayalı Yaklaşım

devam

■ Nesneye dayalı bir dil kullanılarak problem çözülürken, programcı, problemin çözümü için programın hangi fonksiyonlara bölünmesi gerektiğini düşünmek yerine, hangi nesnelere bölünmesi gerektiğini düşünecektir.

■ Nesneye dayalı düşünmek, sadece programın daha kolay oluşturulmasını sağlamayacak, aynı zamanda daha kolay modellenmesini sağlayacaktır. Bu programlama anlamındaki nesne ile gerçek dünyadaki nesne arasındaki yakın ilişkiden kaynaklanmaktadır.

■ Nesneye dayalı programlarda ne tür büyüklükler nesne olarak modellenebilir ?

• Fiziksel Nesneler:

- Asansör kontrol sistemindeki asansör
- Ekonomik modeldeki ülkeler
- Hava trafik kontrol sistemindeki uçaklar
- Bilgisayar kullanıcı arayüzündeki elemanlar
 - Pencereler
 - Menüler
 - Grafik nesneleri (doğru, dikdörtgen, daire)

• Veri yapıları :

- Dizi
- Yığın
- Bağlantılı liste

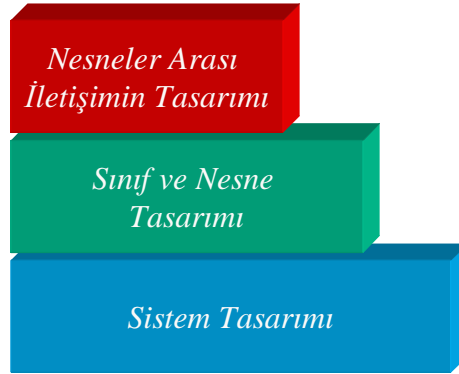
• İnsan kaynakları :

- İşçi
- Öğrenci
- Müşteri

C++ ve NESNEYE DAYALI PROGRAMLAMA

20

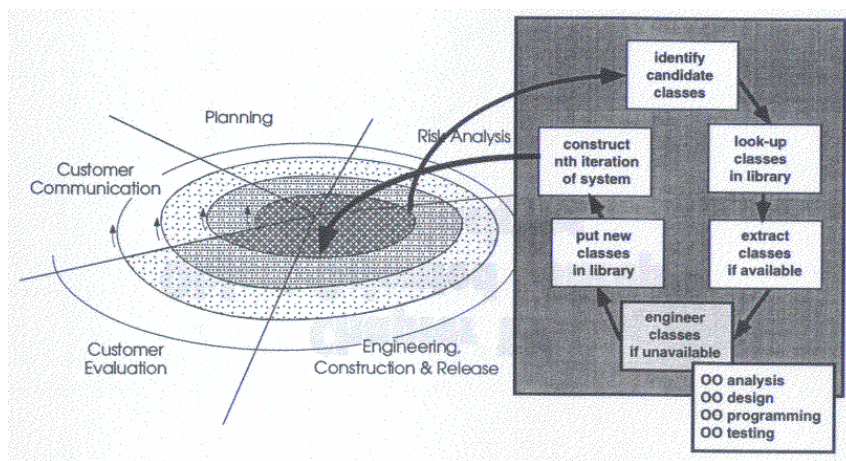
Nesneye Dayalı Tasarım



C++ ve NESNEYE DAYALI PROGRAMLAMA

21

The OO Process Model



C++ ve NESNEYE DAYALI PROGRAMLAMA

22

Nesneye Dayalı Yaklaşımın Avantajları

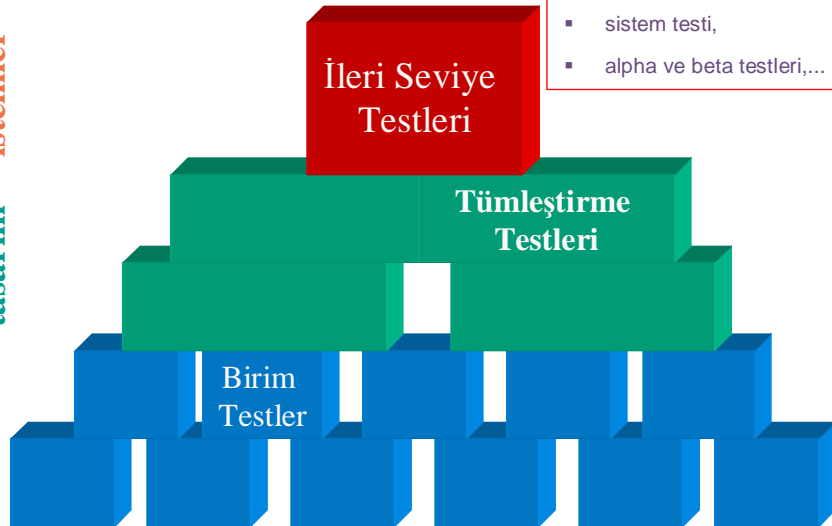
- Okunabilirlik
- Anlaşılabilirlik
- Test ve Bakım
- Tekrar kullanılabilirlik
- Takım Çalışması



C++ ve NESNEYE DAYALI PROGRAMLAMA

23

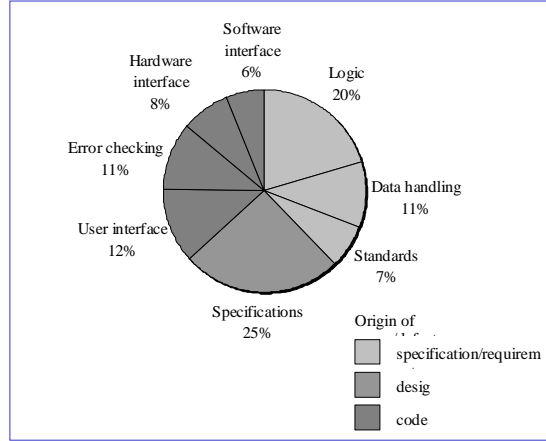
istemler
tasarım
kodlama



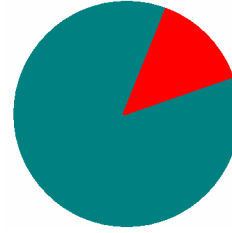
C++ ve NESNEYE DAYALI PROGRAMLAMA

24

Hata Ayıklama



Hatayı düzeltmek için geçen süre



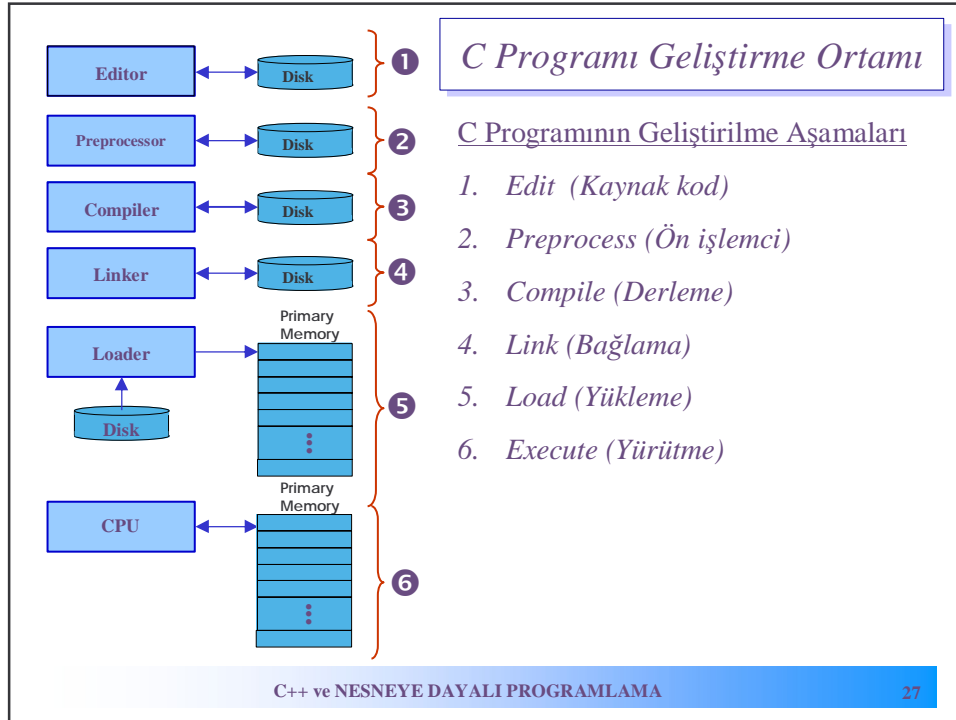
Hatanın türünü ve yerini bulmak için geçen süre

C Programlama Dili

D. Ritchie tarafından iki eski dilden yararlanılarak geliştirilmiştir.

BCPL (Binary Coded Programming Language) ve B

- UNIX işletim sistemini yazmak için kullanılmıştır.
- Günümüzde bir çok işletim sistemi C veya C++ kullanılarak yazılır.
- Donanımdan bağımsız, taşınabilir (portable) programlar yazılabilir.
- C programları fonksiyon (function) adı verilen bağımsız program parçalarından oluşur.
- Programcılar kendi fonksiyonlarını yazabildikleri gibi, C kütüphanesinde bulunan ve önceden yazılmış olan fonksiyonları da kullanabilirler.



```
/* ilk C programı */
#include <stdio.h>
void main()
{
    printf( "Welcome to C!\n" );
}
```

Welcome to C!

- **Açıklamalar**
 - /* ve */ simgeleri arasındaki yazılar göz ardı edilir.
 - Programı anlaşılır hale getirmek için kullanılır.
- **#include <stdio.h>**
 - Önilemci belli bir dosyayı diskten okuyup programa ekler.
 - <stdio.h> dosyasında giriş/çıkış fonksiyonlarının tanımları bulunur.

C++ ve NESNEYE DAYALI PROGRAMLAMA 28

- **void main()**
 - C programları fonksiyonlardan oluşur.
 - İlk çalışan fonksiyonun adı **main** dir.
 - Kıvrıcık parantezler { ve } bir bloğun başını ve sonunu belirtir.
- **printf("Welcome to C!\n");**
 - Bir çıkış fonksiyonudur. Ekrana verilerin yazılmasını sağlar. Bu fonksiyon önceden hazırlanmıştır ve derleyicinin kütüphanesinde yer almaktadır.
 - \ - escape character: printf fonksiyonu için özel işlemler belirler.
 - \n is the newline character: Alt satıra geç.
- ;
 - Komutların sonuna ; konur.

Bellek Kavramı

Bilgisayarların bellekleri sayısal adreslere sahip gözlerden oluşur. Yüksek düzeyli diller ile program yazan programcıların gerçek bellek adreslerini bilmelerine gerek yoktur. Bu nedenle değişkenler tanımlanır. Derleyici bu değişkenler için bellekte yer ayırır.

- Değişkenler bilgisayar belleğindeki gözlere karşı düşerler.
 - Her değişkenin bir adı, boyu, bellek adresi, tipi ve değeri vardır.
 - Değişkene yeni bir değer atandığında bellekteki eski değer kaybolur.
 - Değişkenlerin okunması bellekteki değeri değiştirmez.

Değişken adı	Bellek Gözleri	Adres
sayi1	45	3500
sayi2	12	3501
toplam	57	3502

Değişken Tanımlamaları

Bir C programında kullanılacak olan değişkenlerin programın başında, yürütülen deyimlerden önce tanımlanmaları gerekir.

Değişken tanımlanması,

- bilgisayar belleğinde o değişkenler için yer ayrılmasını sağlar,
- ilgili bellek gözlerinde sembolik isimler verilmesini sağlar.

Değişken İsimleri:

- Harf, rakam ve alt çizgi '_' içerirler.
- Harf veya '_' ile başlamak zorundadır. '_' ile başlaması önerilmez.
- Büyük/küçük harf ayrımı vardır.
- C dilinin anahtar sözcükleri değişken ismi olarak kullanılamaz.
- Özel karakterler ve Türkçe'ye özgü harfler (ş,ğ,ü,ı,ç) kullanılamaz.

Geçerli isimler:

a1, sayi, sayi_3, İlkDeger, SonSayi, alc32, deneme_degeri

Geçersiz isimler:

la1, değer, int

Değişken Tanımlamaları

Programın anlaşılabilirliğini sağlamak için değişkenlere, işlevleriyle ilişkili anlamlı isimler verilmeli.

Birden fazla sözcükten oluşan değişkenlerde ikinci sözcük büyük harf ile başlatılmalı ya da araya '_' konulmalı

Değişken Tipleri:

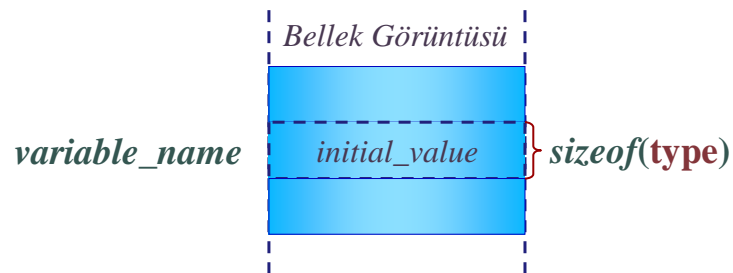
Değişken tipi	Anahtar sözcük	Byte	Çalışma Aralığı
Karakter	char	1	-128 ; 127
Tamsayı	int	2	-32768 ; 32767
Kısa tamsayı	short	2	-32768 ; 32767
Uzun tamsayı	long	4	-2,147,483,648 ; 2,147,483,647
İşaretsiz karakter	unsigned char	1	0 ; 255
İşaretsiz tamsay	unsigned int	2	0 ; 65535
İşaretsiz uzun ts.	unsigned long	4	0 ; 4,294,967,295
Reel sayı	float	4	1.2E-38 ; 3.4E38
Çift hassas. r.s.	double	8	2.2E-308 ; 1.8E308

Değişkenlerin kapladıkları alanlar bazı sistemlerde farklı olabilir.

Değişken Tanımlamaları

C/C++ tip denetimli bir dildir. Bu nedenle program içinde her bir değişken mutlaka bir tip ile ilişkilendirilmelidir:

type *variable_name* = *initial_value* ;



Erim (= Scope) Kavramı

```
{  
    int x = 12;  
    /* only x available */  
    {  
        int q = 96;  
        /* both x & q available */  
    }  
    /* only x available */  
    /* q is out of scope */  
}
```

Operatörler

Operatörleri aldıkları operand sayısına göre üç sınıfta toplayabiliriz:

■ Tekli Operatörler

++, --, !, +, -

■ İkili Operatörler

=, +, -, ==, <, >, *, /

■ Üçlü Operatör ?:

*operatörler
arası öncelik*

()
++ -- + - !
* / %
+ -
< <= > >=
== !=
&&
||
?:
= += -= *= /= %=
,

Önceden/Sonradan Arttırma/Azaltma ++/-- Operatörleri

int a=0,b=2;

a = b++; a=2, b=3

b = --a ; a=1, b=1

a = b-- ; a=1, b=0

b = ++a ; a=2, b=2

Atama Operatörleri

int a,b;

a = a + b;

a += b ;

a = a / b;

a /= b ;

a = a - b;

a -= b ;

a = a * b;

a *= b ;

a = a % b;

a %= b ;

Karşılaştırma Operatörleri

a == b ***Eşitlik***

a != b ***Eşitsizlik***

a > b ***Büyüklik***

a >= b ***Büyük Eşitlik***

a < b ***Küçüklük***

a <= b ***Küçük Eşitlik***

Bit Düzeyinde İşlem Yapan Operatörler

~	DEĞİL
&	VE
/	VEYA
^	Dar VEYA

~=, &=, /=, ^=

Lojik Operatörler

!	Değil
&&	VE
//	VEYA
^^	Dar VEYA

a	b	a&& b
0	0	0
0	nonzero	0
nonzero	0	0
nonzero	nonzero	1

a	b	a b
0	0	0
0	nonzero	1
nonzero	0	1
nonzero	nonzero	1

int a = 0, b = 5, c = -2 ;

((a || b) && c) true

if(a == 5)

if(a = 5)



x	y	⊕
F	F	F
F	T	T
T	F	T
T	T	F

Kaydırma Operatörleri

<<

Sola Kaydırma

>>

Sağa Kaydırma

```
int a = -16 ;
```

```
a = a<<2 ;
```

```
printf(“%d”,a) ;
```

-64

```
a = a>>1 ;
```

```
printf(“%d”,a) ;
```

-32

```
unsigned int a = -16 ;
```

```
a = a<<2 ;
```

```
printf(“%d”,a) ;
```

-64

```
a = a>>1 ;
```

```
printf(“%d”,a) ;
```

2147483616

C++ ve NESNEYE DAYALI PROGRAMLAMA

41

DİZİLER



c[0]	-48
c[1]	6
c[2]	0
c[3]	72
c[4]	1543
c[5]	-89
c[6]	0
c[7]	62
c[8]	-3
c[9]	1
c[10]	6453
c[11]	78

indis değeri

C++ ve NESNEYE DAYALI PROGRAMLAMA

42

ÇOK BOYUTLU DİZİLER

`int b[2][2] = {{1,2},
 {3,4}};`

ikinci boyut

birinci boyut

Başlangıç değerleri satır düzeninde verilmelidir.

<code>b[0][0]</code>	1
<code>b[0][1]</code>	2
<code>b[1][0]</code>	3
<code>b[1][1]</code>	4

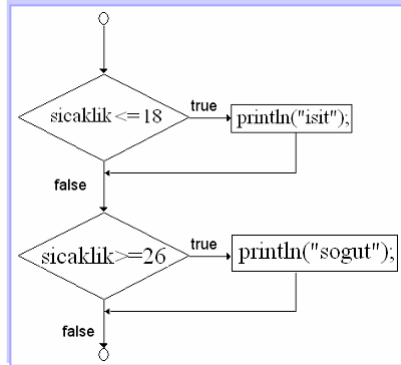
indis değerleri

Kontrol ve Çevrim Yapıları

- ✚ Dallanma Komutları
 - if, if-else, switch
- ✚ Çevrim Oluşturma Komutları
 - for(; ;)
 - while()
 - do{...}while() ;

if, if-else

```
int sicaklik ;  
...  
if ( sicaklik <=18 )  
    printf("isit");  
else  
if ( sicaklik >= 26 )  
    printf("sogut");
```



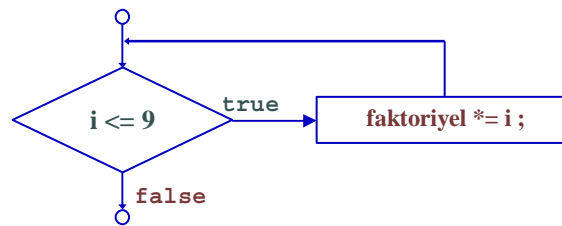
```
printf( sicaklik <= 18 ? "isit" : "sogut" );  
           lojik ifade   true   false
```

```
if ( x > 5 )  
    if ( y > 5 )  
        printf( "x and y are > 5" );  
else  
    printf( "x is <= 5" );
```

```
if ( x > 5 ){  
    if ( y > 5 )  
        printf( "x and y are > 5" );  
}  
else  
    printf( "x is <= 5" );
```

while, do-while döngüsü

```
int faktoriyel = 1, i=1;
while ( i <= 9 ){
    faktoriyel *= i;
    i++;
}
printf("9!=%d",faktoriyel);
```



C++ ve NESNEYE DAYALI PROGRAMLAMA

47

do{...}while(...)

```
int faktoriyel = 1, i=1;
do{
    faktoriyel *= i;
    i++ ;
}while ( i <= 9 );
printf("9!=%d",faktoriyel)
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

48

for döngüsü

```
int faktoriyel, sum, i;

for (i=1, faktoriyel, sum=0; i<=9; sum+=i, i++)
    faktoriyel *= i;

printf("9!=%d", faktoriyel)
```

```
void main() {
    for(int i = 0; i < 100; i++) {
        if(i == 74) break; // Out of for loop
        if(i % 9 != 0) continue; // Next iteration
        printf("%d", i);
    }
    int i = 0;
    while(true) {
        i++;
        int j = i * 27;
        if(j == 1269) break; // Out of loop
        if(i % 10 != 0) continue; // Top of loop
        printf("%d", i);
    } } }
```

Fonksiyonlar

- Kendi içinde bağımsız olarak çalışabilen ve belli bir işlevi yerine getiren program modülleridir.
- C programları bu modüllerden (fonksiyonlar) oluşurlar.
- Fonksiyonların yazılmasındaki temel amaç; büyük boyutlardaki programların daha kolay yazılabilen ve test edilebilen küçük parçalar halinde oluşturulabilmesidir (Böl ve yönet).

Fonksiyonları Özellikleri:

- Her fonksiyonun bir adı vardır. Fonksiyon isimlerinin verilmesinde değişken isimlerinde uygulanan kurallar geçerlidir.
- Fonksiyonlar programın diğer parçalarından etkilenmeden bağımsız bir işlem yapabilirler.
- Belli bir işlevi yerine getirirler. Örneğin, ortalama hesaplamak, ekrana bir veri yazmak, bir dizideki en büyük elemanı bulmak gibi.
- Kendilerini çağıran programdan parametre olarak veri alabilirler.
- Gerektiği durumlarda ürettikleri sonuçları kendilerini çağıran programa parametre olarak geri gönderirler.

Örnek

```
/* Fonksiyon örneği. Küp hesaplayan fonksiyon */
#include <stdio.h>

/* Fonksiyon: kup Bir tamsayının küpünü hesaplar */
long int kup(int x)
{
    long yordimci; // Yerel değişken
    yordimci = x * x * x;
    return yordimci;
}

/* Ana program */
void main()
{
    int giris;
    long int sonuc;
    printf("Bir sayı giriniz: ");
    scanf("%d", &giris);
    sonuc = kup(giris); // Fonksiyon çağırılıyor
    printf("\n%d üssü 3= %ld\n", giris, sonuc);
}
```

Geri Dönüş değerinin tipi

Fonksiyon adı

Giriş parametresi

Yerel değişken.
Sadece fonksiyonun içinde geçerli

Sonuç, çağıran programa
gönderiliyor

Fonksiyon çağırılıyor

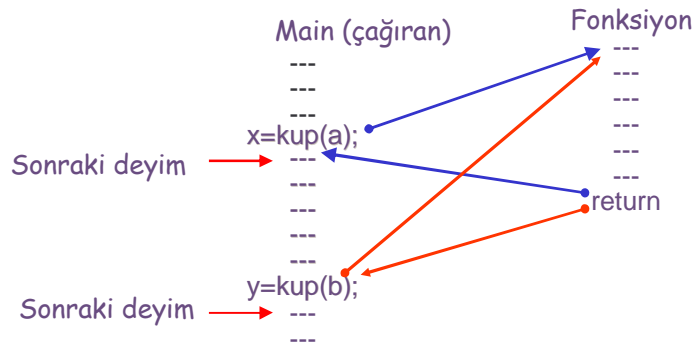
Fonksiyona giden değer

Fonksiyondan gelen değer
yazılacağı bellek gözü

Fonksiyonların Sağladığı Yararlar

- Karmaşık problemler daha küçük parçalara bölünebilir. Her parça ayrı ayrı fonksiyonlar şeklinde çözülerek sonradan ana programda birleştirilebilir.
- Grup çalışmaları için uygun bir ortam hazırlar. Grup elemanları bağımsız fonksiyonları ayrı ayrı tasarlarlar. Son aşamada bu fonksiyonlar ana programda birleştirilir.
- Daha önceden yazılmış fonksiyonlar arşivlerden alınarak kullanılabilir. Aynı program parçasının tekrar yazılmasına gerek kalmaz.
- Programın içinde sık sık tekrar edilen bölümler fonksiyon olarak yazılabilir. Böylece aynı program parçasının defalarca tekrar edilmesine gerek kalmaz.

Fonksiyonların İşleyişi



Fonksiyonların Tanımlanması:

```
geri_dönüş_değeri_tipi  fonksiyon_adı ( parametre listesi )
{
    deyimler
    return <değişken/sabit/ifade>;
}
```

Eğer fonksiyon geriye değer döndürmeyecekse geri dönüş değerinin tipi **void** olarak tanımlanır. Bu durumda fonksiyondan çıkışı sağlayan **return** sözcüğünün yanına bir değer yazılmaz. Bu tür fonksiyonlarda istenirse return sözcüğü yazılmayabilir.

Örneğin: İki tamsayıyı ekrana yazan fonksiyon

```
void yaz(int a, int b)
{
    printf("\nsayı1=%d  sayı2=%d", a, b);
    return;    // Bu satır yazılmayabilir.
}
/** Ana Program (Ana fonksiyon) */
void main()
{
    int i1=450, i2=-90;
    yaz(i,23);
    yaz(18,i2);
    yaz(i1, i2);
}
```

Bir fonksiyonda birden fazla return sözcüğü (çıkış noktası) olabilir.

Örneğin: İki tamsayının büyük olanını bulan ve çağıran programa gönderen fonksiyon.

```
#include <stdio.h>
int buyuk( int a, int b){
    if (a > b) return a;
    else    return b;
}
void main()
{
    int x, y, z;
    printf("\nİki sayı giriniz: ");
    scanf("%d%d", &x, &y);
    z = buyuk(x,y);
    printf("\nDaha büyük olan: %d.", z);
}
```

Fonksiyonların Çağırılması

Fonksiyonlar isimleri yazılarak ve parantez içinde gerekli sayıda argüman gönderilerek çağırılır.

Eğer fonksiyon geriye bir değer döndürüyorsa, bu değer bir değişkene atanabilir, başka bir fonksiyona argüman olarak verilebilir, bir ifadenin içinde kullanılabilir.

Örneğin: Bir tamsayının yarısını hesaplayan ve çağıran programa gönderen fonksiyon.

```
float yarisi( int a)
{
    return (a/2.0);
}
```

Bu fonksiyon aşağıdaki satırlarda gösterildiği gibi çağırılabilir.

```
x=yarisi(5);
z=yarisi(i) + 3*yarisi(k);
printf("\n Sayının yarısı= %f", yarisi(sayi));
f=yarisi((int)yarisi(x));
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

57

Yerel Değişken / Global Değişken:

Fonksiyonların gövdelerinin içinde ({ } arasında) tanımlanan değişkenler sadece o fonksiyonda (main veya diğer) kullanılabilen **yerel değişkenler**dir. O fonksiyon sona erdikten sonra yerel değişkenler bellekten kaldırılırlar. Fonksiyon gövdelerinin dışında tanımlanan (örneğin main'in üstünde) değişkenler ise **global değişkenler**dir. Bu değişkenler bütün fonksiyonlar tarafından yazılıp okunabilirler ve programın çalışması süresince geçerlidirler.

Örneğin; yandaki program parçasında **f** global değişkendir ve tüm fonksiyonlar tarafından kullanılabilir.

yardimci adlı değişken sadece kup adlı fonksiyonda kullanılabilir.

giris adlı değişken ise sadece main içinde kullanılabilir.

```
#include <stdio.h>
float f;           // Global değişken
int long kup(int x)
{
    long yardimci; // kup'e ait yerel değişken
    yardimci = x * x * x;
    return yardimci;
}
/* Ana program */
void main()
{
    int giris;      // main'e ait yerel değişken
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

58

Örnek: Aşağıdaki örnekte aynı global ve yerel değişkenler aynı isimde tanımlanmış (x ve y). Bu durumda fonksiyonun içinde x ve y isimleriyle sadece yerel değişkenlere erişilir.

```
int x = 1, y = 2;           // Global Değişkenler
void demo() {               // Parametre almıyor, değer döndürmüyor
    int x = 88, y = 99;     // yerel değişkenler
    printf("\nFonksiyonun içinde, x = %d y = %d.", x, y);
}
void main() { /**/ Ana program ***/
    printf("\nFonksiyonu çağırılmadan önce, x = %d y = %d.", x, y);
    demo();
    printf("\nFonksiyonu çağırdıktan sonra, x = %d y = %d", x, y);
}
```

Aşağıdaki olumsuz yönlerinden dolayı global değişken kullanımından kaçınmak gerekir:

- Global değişkenler bütün fonksiyonlar tarafından değiştirilebildiği için programdaki hataların ayıklanmasını zorlaştırırlar.
- Grup elemanları arasındaki bağımlılık artar. Hangi global değişkenin ne işlevi olacağına, ismine ve kimin tarafından ne şekilde değiştirileceğine önceden karar vermek gerekir.

Karakter Katarı (= String)

C dilinde, alfabetik sözcükler (string) karakter dizisi şeklinde tanımlanır :

```
char sozcuk[8]; // 7 harflik bir sözcük taşıyabilir.
void main()
{
    sozcuk="merhaba";
    printf("\n Mesaj: %s",sozcuk);
}
```

sozcuk[0]	'm'	• Karakter katarının her elemanı bir karakter içerir.
sozcuk[1]	'e'	
sozcuk[2]	'r'	• Karakter katarlarına ilişkin değerler iki adet çift tırnak (") içinde yazılır.
sozcuk[3]	'h'	
sozcuk[4]	'a'	• Bir karakterlik değerler tek tırnak (') işaretleri arasında yazılır
sozcuk[5]	'b'	
sozcuk[6]	'a'	• '\0' Karakter katarının sona erdiğini belirten özel bir karakterdir.
sozcuk[7]	'\0'	

Örnek

```
/* String: Karakter Dizisi */
#include <stdio.h>
void main()
{
```

```
    char isim[20];
    char mesaj[ ] = "Merhaba";
```

```
    int i;
```

```
    printf(" Adınızı giriniz: ");
```

```
    scanf( "%s", isim );
```

```
    printf( "\n%s %s\n Nasılsın?\n", mesaj, isim );
```

```
    printf("\nHarflerin arasında birer boşluk bırakarak adınız:\n");
```

```
    for ( i = 0; isim[ i ] != '\0'; i++ )
```

```
        printf( "%c ", isim[ i ] );
```

```
    printf( "\n" );
```

```
}
```

20 harflik yer ayrılıyor

8 harflik yer ayrılıyor.
Başlangıç değeri "merhaba"

scanf ile string okunurken
başına & yazılmaz

karakter katarları
ekrana yazılırken %s kullanılır

katarının son harfi: \0

isim katarının i. harfi

Bkz. string.c

C++ ve NESNEYE DAYALI PROGRAMLAMA

61

C++'ın C'ye Getirdiği Gelişmiş Özellikler

- C++, C'nin bir üst kümesidir,
- C'de yazdığınız kodları bir C++ derleyicisi ile derleyebilirsiniz,
- C++'ın nesneye dayalı olmayan özelliklerini C programı yazarken kullanabilirsiniz.

- Açıklama satırları

```
/* This is a comment */
```

```
// This is a comment
```

- C++'da tanımlamayı programın istediğiniz yerinde yapabilirsiniz. Bu programın okunabilirliğini arttıracaktır.

C++ ve NESNEYE DAYALI PROGRAMLAMA

62

C++'ın C'ye Getirdiği Gelişmiş Özellikler devam

```
int a=0;
for (int i=0; i < 100; i++){ // i is declared in for loop
    a++;
    int p=12;                // Declaration of p
    ...                      // Scope of p
}                            // End of scope for i and p
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

63

C++'ın C'ye Getirdiği Gelişmiş Özellikler devam

■ Erim (=scope) Operatörü ::

Kural olarak C'de her değişken tanımlı olduğu blok içerisinde erime sahiptir.

```
int x=1;
void f(){
    int x=2;    // Local x
    x++;        // Local x is 3
}
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

64


```

int x=1;
void f(){
    int x=2;    // Local x
    ::x++;      // Global x is 2
}

```

```

int i=1;
main(){
    int i=2;
    {
        int n=i ;
        int i = 3 ;
        cout << i << " " << ::i << endl ;
        cout << n << "\n" ;
    }
    cout << i << " " << ::i << endl;
    return 0 ;
}

```

```

3 1
2
2 1

```

C++'ın C'ye Getirdiği Gelişmiş Özellikler devam

■ inline fonksiyonlar

C'deki makrolara göre aşağıdaki avantajlara sahiptir

- Hata ayıklama
- Tip denetimi
- Kolay izlenebilir

```
#define sq(x) (x*x)
```

```
inline int SQ(int x){return (x*x); }
```

```
#define max(x,y) (y<x ? x : y)
```

```
inline int max(int x,int y){return (y<x ? x : y); }
```

```
const false = 0 ;  
const double pi = 3.14159 ;  
const double e = 2.71828;  
const M_Size = 100 ;  
const *p=&M_Size ;  
char * const s= "abcde" ;
```

```
#define false 0  
#define pi 3.14159
```

*Tip belirtilmemiş ise
derleyici tipi **int** kabul
eder.*

C++'ın C'ye Getirdiği Gelişmiş Özellikler

devam

- Fonksiyon Parametrelerine Başlangıç Değeri atayabilme

```
int e(int n,int k=2){  
    if(k == 2)  
        return (n*n) ;  
    else  
        return ( mult(n,k-1)*n ) ;  
}
```

e(i+5)
// (i+5)* (i+5)
e(i+5,3)
// (i+5)'in kubu



```
void f(int i, int j=7) ; // dogru  
void g(int i=3, int j) ; // yanlis  
void h(int i, int j=3,int k=7) ; // dogru  
void m(int i=1, int j=2,int k=3) ; // dogru  
void n(int i=2, int j,int k=3) ; // dogru ? yanlis
```

C++'ın C'ye Getirdiği Gelişmiş Özellikler

devam

■ Referans Operatörü &

Bir değişkenin adres bilgisine erişmek için kullanılır.

İki farklı kullanım biçimi vardır:

```
int n ;  
int& nn = n ;  
double a[10] ;  
double& last = a[9] ;  
const char& new_line = '\n' ;
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

71

■ İki değişkenin içeriğini takas eden fonksiyon : swap()

```
void swap(int a, int b){
```

```
    int temp = a ;
```

b=5

```
    a = b ;
```

a=3

```
    b = temp ; }
```

GDA

```
main(){
```

```
    int i=3,j=5 ;
```

```
    swap(i,j) ;
```

```
    cout << i << " " << j << endl ;
```

```
}
```

GDA

5

3

sistem yığını

bellek

i	3
j	5
	...
a	3
b	3

35

C++ ve NESNEYE DAYALI PROGRAMLAMA

72

```

void swap(int *a, int *b){
    int temp = *a ;
    *a = *b ;
    *b = temp ; }

```

bellek

i	3
j	5
	...
a	adr
b	adr

5 3

```

main(){
    int i=3,j=5 ;
    swap(&i,&j) ;
    cout << i << " " << j << endl ;
}

```

C++ ve NESNEYE DAYALI PROGRAMLAMA
73

```

void swap(int& a,int& b){
    int temp = a ;
    a = b ;
    b = temp ; }

```

3 5

```

main(){
    int i=3,j=5 ;
    swap(i,j) ;
    cout << i << " " << j << endl ;
}

```

C++ ve NESNEYE DAYALI PROGRAMLAMA
74

```
void shift(int& a1,int& a2,int& a3,int& a4){
    int tmp = a1 ;
    a1 = a2 ;
    a2 = a3 ;
    a3 = a4 ;
    a4 = tmp ;
}
```

```
main(){
    int x=1,y=2,z=3,w=4;
    cout << x << y << z << w << endl;
    shift(x,y,z,w) ;
    cout << x << y << z << w << endl;
    return 0 ;
}
```

```
int squareByValue(int a){
    return (a*a) ;
}
```

```
void squareByReference(int& a){
    a *= a ;
}
```

```
main(){
    int x=2,y=3,z=4 ;
    squareByPointer(&x) ;
    cout << x << endl ;
    squareByReference(y) ;
    cout << y << endl ;
    z = squareByValue(z) ;
    cout << z << endl ;
}
```

```
void squareByPointer(int *aPtr){
    *aPtr *= *aPtr ;
}
```

4
9
16

C++'ın C'ye Getirdiği Gelişmiş Özellikler

■ Dinamik Bellek Kullanımı

C'de dinamik bellek kullanımı standart kütüphaneler kullanılarak gerçekleştirilmektedir:

```
int *p ; p = (int *) malloc(N*sizeof(int)) ; free(p) ;
```

■ C++'da iki yeni operatör : **new** ve **delete**

```
int *p ;
```

```
p = new int[N] ;
```

```
delete []p ;
```

```
int *p,*q ;  
p = new int[9] ;  
q = new int(9) ;
```



■ İki boyutlu matris

```
❶ double ** q ;
```

```
q = new double*[row] ; // rowxcolumn'lik bir matris
```

```
for(int i=0;i<row;i++)
```

```
    q[i] = new double[column] ;
```

```
.....
```

```
for(int i=0;i<row;i++)
```

```
    delete q[i] ;
```

```
delete []q ;
```

i. satır j. sütun elemanına erişim : **q[i][j]**

■ İki boyutlu matris

```
2 double ** q,*t ;  
  p = new double*[row] ; // rowxcolumn'lik bir matris  
  t = new double[row*column] ;  
  for(int i=0,col=0;i<row;i++,col+=column)  
    q[i] = t + col ;  
  .....  
  delete q[0] ;  
  delete q ;  
  
  i. satır j. sütun elemanına erişim : q[i][j]
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

79

```
double ** q,*t ;  
memoryAlign = column % 4 ;  
memoryWidth = ( memoryAlign == 0 ) ?  
                column : (column+4 -memoryAlign) ;  
p = new double*[row] ; // rowxmemoryWidth'lik bir matris  
t = new double[row*memoryWidth] ;  
for(int i=0,col=0;i<row;i++,col+=memoryWidth)  
  q[i] = t + col ;  
.....  
delete q[0] ;  
delete q ;
```



C++ ve NESNEYE DAYALI PROGRAMLAMA

80

C++'ın C'ye Getirdiği Gelişmiş Özellikler

devam

■ Fonksiyon Yükleme (=Function Overloading)

```
double average(const double a[],int size) ;
```

```
double average(const int a[],int size) ;
```

```
double average(const int a[], const double b[],int size) ;
```

```
❶ double average(const int a[],int size) {  
    double sum = 0.0 ;  
    for(int i=0;i<size;i++) sum += a[i] ;  
    return ((double)sum/size) ;  
}
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

81

```
❷ double average(const double a[],int size) {  
    double sum = 0.0 ;  
    for(int i=0;i<size;i++) sum += a[i] ;  
    return (sum/size) ;  
}
```

```
❸ double average(const int a[],const double b[],int size) {  
    double sum = 0.0 ;  
    for(int i=0;i<size;i++) sum += a[i] + b[i] ;  
    return (sum/size) ;  
}
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

82

```

main() {
    int    w[5]={ 1,2,3,4,5 } ;
    double x[5]={ 1.1,2.2,3.3,4.4,5.5 } ;

    ❶ cout << average(w,5) ;
    ❷ cout << average(x,5) ;
    ❸ cout << average(w,x,5) ;
    return 0 ;
}

```

C++'ın C'ye Getirdiği Gelişmiş Özellikler devam

■ Function Templates

```

template <class T>
void printArray(T *array, const int size){
    for(int i=0; i < size; i++)
        cout << array[i] << " ";
    cout << endl ;
}

```

```

main() {
    int    a[3]={ 1,2,3} ;
    double b[5]={ 1.1,2.2,3.3,4.4,5.5} ;
    char   c[7]={ 'a', 'b', 'c', 'd', 'e' , 'f', 'g' } ;
    ❶ printArray(a,3)      ;
    ❷ printArray(b,5)      ;
    ❸ printArray(c,7)      ;
    return 0 ;
}

```

```

void printArray(int *array,cont int size){
    for(int i=0;i < size;i++)
        cout << array[i] << " , " ;
    cout << endl ;
}

void printArray(char *array,cont int size){
    for(int i=0;i < size;i++)
        cout << array[i] ;
    cout << endl ;
}

```

İşlev Yükleme

C++'da yerleşik operatörlere (+, -, = ve ++ gibi) aldıkları operandların tipine göre yeni işlevler yüklenebilir. Aşağıda örnek olarak İki karmaşık sayıyı toplamak için + operatörüne yeni işlev yüklenmiştir:

```
struct ComplexT{
    float real,img;
};
ComplexT operator+(ComplexT v1, ComplexT v2){
    ComplexT result;
    result.real=v1.real+v2.real;
    result.img=v1.img+v2.img;
    return result;
}
void main(){
    ComplexT c1,c2,c3;
    c3=c1+c2; // c1+(c2)
}
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

87

C++'ın C'ye Getirdiği Gelişmiş Özellikler

```
#include <string>
#include <iostream>
using namespace std;
int main() {
    string test;
    while(test.empty() || test.size() <= 5)
    {
        cout << "Type a string longer string. " << endl;
        cin >> test;
    }
    printf("%s",s.c_str())
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

88

```

#include <iostream>
namespace F {
    float x = 9;
}
namespace G {
    using namespace F;
    float y = 2.0;
    namespace INNER_G {
        float z = 10.01;
    }
}

int main(void) {
    float x = 19.1;
    using namespace G;
    using namespace G::INNER_G;
    std::cout << "x = " << x << std::endl;
    std::cout << "y = " << y << std::endl;
    std::cout << "z = " << z << std::endl;
    return 0;
}

```

Nesneye Dayalı Programlama Kavramları

■ Fonksiyonel programlamada probleme

“Verilen problemin çözümü için **algoritmayı** hangi **fonksiyon**lara parçalamalıyım” ?

sorusu ile yaklaşılır.

■ Nesneye dayalı programlamada ise

“Verilen problemin çözümü için **veriyi** hangi **nesnelere** parçalamalıyım” ?

sorusu ile yaklaşılır. Nesneye dayalı programlamada temel tasarım elemanı “nesneler”dir.

Nesneye Dayalı Programlama Kavramları

■ Nesneye dayalı bir dil kullanılarak problem çözülürken, programcı, problemin çözümü için programın hangi fonksiyonlara bölünmesi gerektiğini düşünmek yerine, hangi nesnelere bölünmesi gerektiğini düşünecektir.

■ Nesneye dayalı düşünmek, sadece programın daha kolay oluşturulmasını sağlamayacak, aynı zamanda daha kolay modellenebilmesini sağlayacaktır. Bu programlama anlamındaki nesne ile gerçek dünyadaki nesne arasındaki yakın ilişkiden kaynaklanmaktadır.

■ İnsanlar olayları zihinlerinde nesneler biçiminde canlandırır.

Örnek : Bir binanın asansör sistemi

- **currentFloorNumber**
- **numberOfPassengersAboard**
- **listOfButtonsPushed**

durum



C++'da Sınıf (=Class) Yapısı

```
struct Time{  
    int hour ;    // 0-23  
    int minute;  // 0-59  
    int second;  // 0-59  
}  
fonksiyonlar ?
```

veri

```
class Time{  
public:  
    Time();  
    void SetTime(int,int,int) ;  
    void PrintTime() ;  
private:  
    int hour ;    // 0-23  
    int minute;  // 0-59  
    int second;  // 0-59  
}
```

davranış

durum

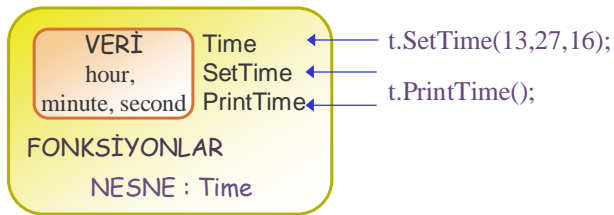
C++ ve NESNEYE DAYALI PROGRAMLAMA

93

```
void Time::Time(){ hour=minute=second=0;}  
  
void Time::SetTime(int h,int m,int s){  
    hour   = (h>=0 && h<24) ? h : 0 ;  
    minute = (m>=0 && m<60) ? m : 0 ;  
    second = (s>=0 && s<60) ? s : 0 ;  
}  
  
void Time::PrintTime(){  
    cout << ((hour == 0) || (hour == 12) ? 12 : hour%12)  
        << ":" << (minute < 10) ? "0" : "" << minute  
        << ":" << (second < 10) ? "0" : "" << second  
        << (hour < 12 ? " AM" : " PM") ;  
}
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

94



```
void main(){
    Time t ;

    t.PrintTime();
    t.SetTime(13,27,16) ;
    t.PrintTime();
    t.SetTime(99,99,99) ;
    t.PrintTime();
    cout << endl ;
    return 0 ;
}
```

12:00:00 AM
1:27:16 PM
12:00:00 AM

Nesne İşaretçileri

```
void main(){
    Time *t ;
    t = new Time ;
    t->PrintTime();
    t->SetTime(13,27,16) ;
    t->PrintTime();
    t->SetTime(99,99,99) ;
    t->PrintTime();
    cout << endl ;
    return 0 ;
}
```

12:00:00 AM
1:27:16 PM
12:00:00 AM

C++ Terminolojisi

- Sınıf : veriler ve o veriler üzerinde işlem yapan fonksiyonlar grubudur. C'deki struct yapısına oldukça benzer. Program içinde bir değişken için kullanılabilecek yeni bir tip oluşturmaktadır.
- Nesne : Belirli bir tipten yaratılan değişken gibi belirli bir sınıftan yaratılan bir kopyadır. Program doğrudan nesneler üzerinde işlem yapar.
- Metod : Nesneye dayalı programlamada, bir sınıf içinde tanımlanan üye fonksiyonlara metod adı verilmektedir.
- Mesaj : Nesneye dayalı programlamada bir sınıfa ait bir üye fonksiyonunun çağırılmasına nesneye dayalı programlamada mesaj denir.

C++ ve NESNEYE DAYALI PROGRAMLAMA

97

class Time { int hour; int minute; int second ; public: void SetTime(int h,int m,int s){hour=h;minute=m;second=s} void PrintTime(); };	Sınıf Tanımı Nitelikler 1. metod 2. metod
---	--

```
void Time::PrintTime()
{
cout << ((hour == 0) || (hour == 12) ? 12 : hour%12)
<< ":" << (minute < 10) ? "0" : "" << minute
<< ":" << (second < 10) ? "0" : "" << second
<< (hour < 12 ? " AM" : " PM");
}
```

2. Metodun gövdesi

void main() { Time t; t.SetTime(13,26,6); t.PrintTime(); gönderiliyor }	t nesnesi yaratıldı nesneye bir mesaj gönderiliyor nesneye bir başka mesaj gönderiliyor
---	--

C++ ve NESNEYE DAYALI PROGRAMLAMA

example1.cp

98

```
class Time {
    int hour;
    int minute;
    int second ;
public:
    void SetTime(int h,int m,int s){hour=h;minute=m;second=s;} ;
    void PrintTime();
};
```

Sınıf tanımı
Nitelikler

1. metod
2. metod

```
void main()
{
    Time t;
    t.hour = 13 ;
    t.minute = 26 ;
    t.second = 6 ;
    t.PrintTime();
}
```

Error:
"Time::hour is not accessible"
"Time::minute is not accessible"
"Time::second is not accessible"

Nesne Üyelerine Erişimin Denetlenmesi

■ Public

Public üyelere programdaki herhangi bir fonksiyon tarafından erişilebilir.

■ Private

■ Herhangi bir tanımlama yok ise varsayılan tanımlama private'dir.

■ Private üyeler sadece o sınıfa ait üyeler veya o sınıfın friend üyeleri tarafından erişilebilir.

■ Protected (Public ile Private arasında)

Private üyeler sadece o sınıfa ait üyeler, o sınıftan türetilmiş sınıfların üyeleri veya o sınıfın friend üyeleri tarafından erişilebilir.

```

class Point {
public:
    void SetPoint(float, float);    // set coordinates
    float GetX() const { return x; }; // get x coordinate
    float GetY() const { return y; }; // get y coordinate
private:    // accessible by derived classes
    float x, y;    // x and y coordinates of the Point
};
void Point::SetPoint(float a,float b){ x=a; y=b; }

```

```

main(){
    Point p;
    p.x=1;    //p.SetPoint(1,2);
    p.y=2;
}

```

```

class Time {
    int hour;
    int minute;
    int second ;
public:
    // Get Functions
    void SetTime(int h,int m,int s){hour=h;minute=m;second=s;};
    void SetHour(int h){hour= (h>=0 && h<24) ? h : 0;};
    void SetMinute(int m){minute= (m>=0 && m<60) ? m : 0;};
    void SetSecond(int s){second= (s>=0 && s<60) ? s : 0;};
    // Get Functions
    int  GetHour(){return hour;};
    int  GetMinute(){return minute;};
    int  GetSecond (){return second;};
    void PrintTime();
};

```

Bir sınıfın **friend** fonksiyonları ve sınıfları

■ Bir fonksiyon yada bir sınıf bir başka sınıfın **friend**'i olarak tanımlanabilir. Bu durumda **friend** olarak tanımlanan fonksiyon yada sınıfın üyeleri o sınıfın tüm üyelerine erişebilir.

```
class A{
    friend class B;    // Class B is a friend of class A
private:
    int i;
    float f;
public:
    void fonk1(char *c);
};
class B{
    int j;
public:
    void fonk2(A s){ cout << s.i << endl ;} // B can access private members of A
};
```

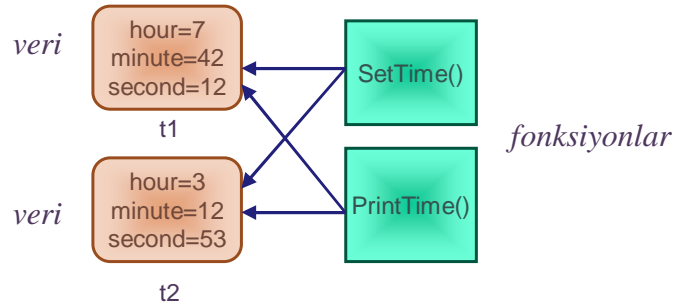
```
class ComplexT{
    friend void print(ComplexT); // print function is a friend of ComplexT
    float real,img; // private
    :
};

void print(ComplexT z)    // is not a member of ComplexT but a friend
{
    cout << z.real << " + i " << z.img;
}
```

Özel bir işaretçi : **this**

■ Nesne = Veri + Fonksiyon

■ Her nesnenin veri alanı bellekte farklı alana yerleştirilmiştir. Ancak fonksiyonlar için bir kez yer alınır ve o sınıftan tüm nesneler aynı fonksiyon alanını kullanırlar.



Bu durumda fonksiyon hangi nesnenin verisini işleyeceğini nereden biliyor ?

C++ ve NESNEYE DAYALI PROGRAMLAMA

105

Bir üye fonksiyon çağrıldığında, nesnenin adresini taşıyan **this** adlı özel bir işaretçi çağrılan fonksiyona parametre olarak aktarılır.

```
class dlink{
    dlink *previous;
    dlink *next;
public:
    void insert(dlink *); // inserts a new node into the list
};
void dlink::insert(dlink * p)
{
    p-> next = next;
    p-> previous =this;
    next -> previous =p;
    next =p;
}
:
dlink dl1,dl2;
:
dl1.insert(&dl2);
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

106

Giriş/Çıkış : <iostream.h>


C++ programınıza “#include <iostream.h>” satırı eklerseniz dört tane nesne yaratılır:

cin

("see-in" okunur) standart giriş cihazından (tuş takımı) değer okur.

cout

("see-out" okunur) standart çıkış cihazına (ekran) değer yazar.

 **cerr** ("see-err" okunur) standart hata cihazına (ekran) tamponlanmamış hataları yazar.

 **clog** ("see-log" okunur) standart hata cihazına (ekran) tamponlanmış hataları yazar.

Ekrana bir değer yazmak için, **cout**'u izleyen << operatörünü kullanmanız gerekir:

```
#include<iostream.h>
void main() {
    int i=5;
    float f=4.6;
    cout << "Integer Number = " << i << " Real Number=" << f;
}
```

Klavyeden bir değer okumak için ise, **cin**'i izleyen >> operatörünü kullanmanız gerekir:

```
#include<iostream.h>
void main() {
    int i,j;
    cout << "Give to Numbers" << endl ;
    cin >> i >> j;
    cout << "Sum= " << i + j << "\n";
}
```

Nesnelere Başlangıç Değeri Verilmesi : Kurucu Fonksiyonlar

- ✚ Sınıflar, kurucu (= constructor) fonksiyon adı verilen özel bir üye fonksiyona sahiptirler.
- ✚ Kurucu fonksiyonun adı sınıf adı ile aynıdır.
- ✚ Kurucu fonksiyonlar parametre alabilirler, ancak bir değer döndürmezler.
- ✚ Kurucu fonksiyonlar, bir nesne yaratıldığında doğrudan derleyici tarafından çağrılır. Amaç sınıf üyelerine başlangıç değerlerinin verilmesidir.

```
class ComplexT{  
    float real,img;  
public:  
    ComplexT(){    // kurucu fonksiyon  
        real=0;  
        img=0;  
    }  
};
```

```
void main()  
{  
    ComplexT z1,z2; // kurucu iki defa cagrilir  
    ComplexT *zp = new ComplexT; // kurucu bir kez cagrilir  
}
```

```
class ComplexT{
    float real,img;
public:
    ComplexT(float r){real=r; img=0;} ;
    ComplexT(float r,float i){real=r; img=i;};
    :    // diger uye fonksiyonlar
};
```

```
void main()
{
    ComplexT z1(0.3);
    ComplexT z2(0.5 , 1.2);
    ComplexT *zp=new ComplexT(0.4);
    ComplexT z3; // Hata!!!
    //Could not find a match for ComplexT:: ComplexT()
}
```

Kopyalayıcı Kurucular

🚦 Kopyalayıcı kurucu fonksiyonlar, o sınıftan bir nesnenin üyelerini yeni bir nesneye kopyalamakta kullanılır. Bu nedenle parametrelerden biri aynı sınıftan bir nesnedir.

🚦 Eğer programcı tarafından tanımlanmamış ise derleyici bir tane yaratır. Yaratılan kurucu fonksiyon nesnenin birebir kopyasını oluşturur.

```
class string{
    int size;
    char *contents;
public:
    string();           // default constructor
    string(string &);  // copy constructor
    void set (int, char *); // An ordinary member function
    void print();
};
```



```

string::string()      // Kurucu fonksiyon
{
    size = 0; contents = new char[1];
    strcpy(contents, "");
}
string::string(string &in_object) // Kopyalayıcı kurucu fonksiyon
{
    size = in_object.size;
    contents = new char[strlen(in_object.contents)];
    strcpy(contents, in_object.contents);
}
void string::set(int in_size, char *in_data)
{
    size = in_size;
    delete contents; contents = new char[strlen(in_data)];
    strcpy(contents, in_data);
}
void string::print()
{
    cout<< contents << " " << size << endl;
}

```

example4.cpp
example5.cpp

```

void main()
{
    string my_string;
    my_string.set (8, "string 1");
    my_string.print();
    string other=my_string; // Copy constructor is invoked
    string more(my_string); // Copy constructor is invoked
    other.print();
    more.print();
}

```

Nesne Dizilerine Başlangıç Değeri Verilmesi

✚ Nesne dizilerinde kurucu fonksiyon her dizi elemanı için teker teker çağrılır:

```
class ComplexT{
    float real, img;
public:
    ComplexT(float, float);
};
ComplexT::ComplexT(float d1, float d2=1){
    real = d1; img = d2;
}

void main(){
    ComplexT s[ ]={ {1.1}, {3.3}, ComplexT(4.4,1.1)};
}
```

Yokedici (= Destructor) Fonksiyonlar

✚ Yokedici fonksiyonlar, kurucu fonksiyonlar gibi o sınıftan bir nesnenin üyelerini yeni nesneye kopyalamakta kullanılır. Bu nedenle parametrelerden biri aynı sınıftan bir nesnedir.

✚ Yokedici fonksiyonun adı ~ ile başlar ve sınıf adı ile aynıdır.

✚ Yokedici fonksiyonlar parametre almazlar ve bir değer döndürmezler.

✚ Yokedici fonksiyonlar bir kere ve otomatik olarak nesne erimi dışına çıkıldığında derleyici tarafından çağrılır.

```

class string{
    int size;
    char *contents;
public:
    string();           // kurucu fonksiyon
    string(string &);   // kopyalayıcı kurucu fonksiyon
    void set (int, char *); // An ordinary member function
    void print();
    ~string();          // Yokedici fonksiyon
};

```

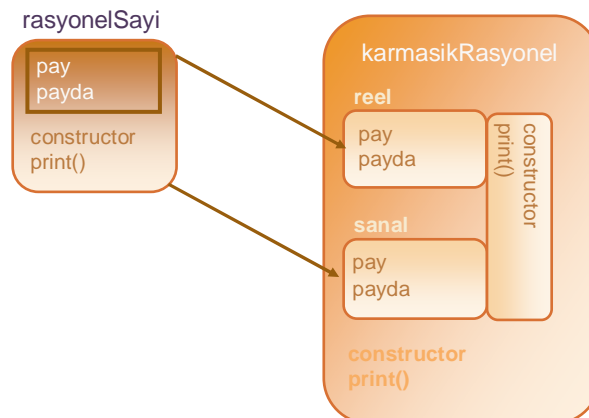
```

string::~~ string()
{
    delete contents;
}

```

İççe Sınıflar

- ✚ Bir sınıf üye olarak başka bir sınıftan nesne içerebilir.



```

class rasyonelSayi{    // rasyonel sayilari modelleyen sinif tanimi
    int pay,payda;
public:
    rasyonelSayi(int, int);
    void print();
};

rasyonelSayi::rasyonelSayi(int py, int pyd) // kurucu fonksiyon
{
    pay=py;
    if (pyd==0) payda=1;
    else payda=pyd;
}
void rasyonelSayi::print()
{
    cout << pay << "/" << payda;
}

```

C++ ve NESNEYE DAYALI PROGRAMLAMA

119

```

class karmasikRasyonel { // rasyonel sayilar
    rasyonelSayi reel,sanal ;    // nesne uyeler
public:
    karmasikRasyonel(int,int); // kurucu fonksiyon
    void print();
};

karmasikRasyonel::karmasikRasyonel(int r,int i):real(r,1),img(i,1)
{
    :
}
void karmasikRasyonel::print()
{
    reel.print();
    sanal.print();
}
void main() {
    karmasikRasyonel karmasik(2,5);
    karmasik.print();
}

```

example6.cpp

Üye nesnelere başlangıç
değerleri veriliyor

C++ ve NESNEYE DAYALI PROGRAMLAMA

120

Sabit Nesneler ve **const** Üye Fonksiyonları

const, değiştirilemez nesneler tanımlamak için kullanılır. Const ile tanımlı bir nesnenin herhangi bir üyesini değiştirmeye çalıştırmak hata oluşturacaktır.

```
const ComplexT cz(0,1); // Sabit nesne
```

Bazen bir sınıfın belirli bazı üyelerinin değiştirilmesini önlemek isteriz. Bu durumda ilgili üye fonksiyonun sonuna const belirteci konur.

```
void print() const // sabit fonksiyon
{
    cout << "complex number= " << real << ", " << img;
}
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

121

example7.cpp

```
class ComplexT{
    float real,img;
public:
    ComplexT(float, float); // kurucu fonksiyon
    void print() const; // sabit fonksiyon
    void reset() {real=img=0;} // sabit degil
};

ComplexT::ComplexT(float r=0,float i=0){
    real=r;
    img=i;
}

void ComplexT::print() const { // sabit fonksiyon
    cout << "complex number= " << real << ", " << img;
}
```

```
void main() {
    const ComplexT cz(0,1); // sabit nesne
    ComplexT ncz(1.2,0.5) // sabit nesne degil
    cz.print(); // OK
    cz.reset(); // HATA !!!
    ncz.reset(); // OK
}
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

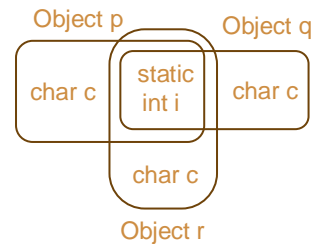
122

static Sınıf Üyeleri

Genel olarak bir sınıfa ait nesnelerin verileri bellekte farklı bölgelerde yer alır. Ancak bazı durumlarda, belirli bir üyenin ortak bir alanda tek bir kopyasının bulunması isteyebiliriz. Bu durumda static anahtar sözcüğünden yararlanıyoruz. Static tanımlı üyeler public yada private tanımlı olabilirler. Static tanımlı üyeler o sınıftan hiçbir nesne var olmasa bile bellekte yer alırlar. Bu durumda static üyeye erim operatörü ile erişilir : **A::i = 5 ;**

```
class A{
...
public:
    static SetVal(int j){i=j;}
};
```

```
class A{
    char c;
    static int i;
};
void main()
{
    A p,q,r;
    :
}
```



```
class A{
    char c;
    static int count; // Number of created objects (static data)
public:
    static void GetCount(){return count;} // Static function to initialize number
    A(){count++; cout<< endl << "Constructor " << count;}
    ~A(){count--; cout<< endl << "Destructor " << count;}
};
int A::number=0; // Allocating memory for number
```

```
void main(){
    cout<<"\n Entering 1. BLOCK.....";
    A a,b,c;
    {
        cout<<"\n Entering 2. BLOCK.....";
        A d,e;
        cout<<"\n Exiting 2. BLOCK.....";
    }
    cout<<"\n Exiting 1. BLOCK.....";
}
```

```
Entering 1. BLOCK.....
Constructor 1
Constructor 2
Constructor 3
Entering 2. BLOCK.....
Constructor 4
Constructor 5
Exiting 2. BLOCK.....
Destructor 5
Destructor 4
Exiting 1. BLOCK.....
Destructor 3
Destructor 2
Destructor 1
```

İşlev Yükleme

- C tip-duyarlı ve -odaklı bir dildir. Her operatör belirli tiplerde operand alır.
- C'de temel tiplerden ve türetilmiş tiplerden yeni tipler türetilir.
- Türetilen tipler için mevcut operatörleri kullanabilir miyiz ?
- C'de operatörleri aldıkları operand sayılarına göre

🚩 Tekli Operatörler

-, +, !, ++, --, new, delete, sizeof()

🚩 İkili Operatörler

+, -, *, /, %, ^, &, |, ~, ==, <, >, +=, -=, *=, /=, %=, ^=, &=, |=, <<, >>, <<=, >>=, <=, >=, &&, ||, ->*, ->, [], (), delete, ::, .

🚩 Üçlü operatör ?:

şeklinde sınıflandırabiliriz.

example9.cpp

```
/* A class to define complex numbers */
class ComplexT{
    float real,img;
public:
    : // Member functions
    ComplexT operator+(ComplexT&); // header of operator+ function
};

/* The Body of the function for operator + */
ComplexT ComplexT::operator+(ComplexT& z)
{
    ComplexT result;
    result.real = real + z.real;
    result.img = img + z.img;
    return result;
}

void main()
{
    ComplexT z1,z2,z3;
    : // Other operations
    z3=z1+z2;           like    z3 = z1.operator+(z2);
}
```


İşlev Yükleme

devam

Yerel olarak geçici nesne yaratmaktan kaçının.

Aşağıdaki fonksiyon daha az bellek kullanır ve daha hızlıdır.

Neden?

ComplexT operator+(const ComplexT&); // header of operator+ function

```
/* The Body of the function for operator + */
ComplexT ComplexT::operator+(const ComplexT& z)
{
    float myReal,myImg ;
    myReal = real + z.real;
    myImg = img + z.img;
    return ComplexT(myReal,myImg);
}
```

example10.cpp

Atama İşlevi (=) Yükleme

Genellikle bir sınıfın birkaç üyesini kopyalamak istediğimizde atama operatörüne işlev yükleriz :

```
void ComplexT::operator=(const ComplexT& z)
{
    real = z.real;
    img = z.img;
}
```

```

class string{
    int size;
    char *contents;
public:
    void operator=(const string &s);      // assignment operator
    :    // Other methods
};

void string::operator=(const string &s)
{
    size = s.size;
    contents = new char[strlen(s.contents)];
    strcpy(contents, s.contents);
}

```

Atama İşlevi (=) Yükleme

devam

Bir önceki örnekteki atama operatörü ile

`s1=s2=s3;`

şeklinde iç içe atamalar yapılamaz. Çünkü işlev yüklediğimiz atama operatörü aynı sınıftan bir nesne döndürmemektedir.

example11.cpp

```

string & string::operator=(const string &s)
{
    size = s.size;
    contents = new char[strlen(s.contents)];
    strcpy(contents, s.contents);
    return *this;
}

```

Tekil İşlevler Yükleme

🚦 Tekli Operatörler : Tek bir operand alırlar

-, +, !, ++, --

Bu nedenle herhangi bir değer döndürmezler.

```
void ComplexT::operator++()
{
    re=re+0.1;
}

void main()
{
    ComplexT z(1.2, 0.5);
    ++z;
    z.print();
}
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

133

Tekil İşlevler Yükleme

devam

Eğer tekli operatörlerin bir değer döndürmesini istersek, o sınıftan bir nesne geri döndürecek şekilde aşağıdaki gibi prototip fonksiyon değiştirilmelidir:

example12.cpp

```
ComplexT & ComplexT::operator++()
{
    real=real+0.1;
    return *this;
}

void main()
{
    ComplexT z1(1.2, 0.5),z2;
    z2= ++z1;           // Prefix or postfix
    z2.print();
}
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

134

Sınıf Yapısı

Kalıtım

Çok Şekillilik

KALITIM (= Inheritance)

C++'ın yazılan kodun yeniden kullanılabilir olmasını sağlayan mekanizması kalıttır. Yeniden kullanılabilirlikten, bir sınıfın alınıp bir başka yazılım uygulamasında da kullanılabilmesini anlıyoruz. Bu özellik yazılım geliştirme çevrimini kısaltırken aynı zamanda yazılımın daha gürbüz olmasını sağlayacaktır.

Tarihçe :

- Kopyala ve Yapıştır + Uyarla + Hata Ayıkla,
- Tekrar tekrar kullanılan fonksiyonlar için kütüphaneler oluştur,
- Yeni yazılım projesi ≡

Kütüphane Fonksiyonları (Uyarla + Hata Ayıkla)

KALITIM

Çözüm :

- Sınıf Kütüphaneleri

Problemleri daha iyi modellediklerinden yeni bir proje için kullanılmak istenildiklerinde daha az değiştirilme ihtiyacı duyarlar.

- C++ bir sınıfın kodunu değiştirmeden eklentiler yapmamıza olanak tanır. Bu kalıtım yolu ile bir temel sınıftan yeni bir sınıf türetilmesi şeklinde olur. **türetilen sınıf** ile **temel sınıf** arasında “is a” şeklinde bir hiyerarşik ilişki söz konusudur.

- Bir temel sınıftan türetilen sınıfı belirtmek için türetilen sınıf adından sonra “:” konup temel sınıf adı yazılır.

KALITIM

```
class teacher { // temel sınıf
public:
    char *Name;
    int Age,numberOfStudents;
    void setName (char *new_name){Name=new_name;}
};
```

```
class principal : public teacher { // türetilmiş sınıf
    char *schoolName;
    int numberOfTeachers;
public:
    void setSchool(char *s_name){schoolName=s_name;}
};
```

principal **is a** teacher

temel sınıf ile türetilmiş
sınıf arasında **is a** ilişkisi
vardır

principal (türetilmiş sınıf)

teacher (temel sınıf)

Name,
Age,
numberOfStudents
setName(char *)

SchoolName
numberOfTeachers
setSchool(char *)

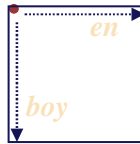
void main()

```
{  
    teacher t1;  
    principal p1;  
    p1.setName(" Principal 1");  
    t1.setName(" Teacher 1");  
    p1.setSchool(" Elementary School");  
}
```

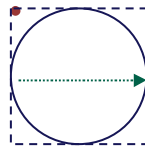
C++ ve NESNEYE DAYALI PROGRAMLAMA

139

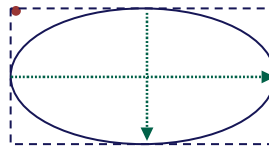
• nokta



dikdörtgen



çember



elips

C++ ve NESNEYE DAYALI PROGRAMLAMA

140

```

class point { // temel sınıf
    int x,y;
public:
    void setPoint(int X,int Y){x=X;y=Y;}
};

```

```

class rectangle : public point { // türetilmiş sınıf
    int Width,Height ;
public:
    void setSize(int w,int h){Width=w;Height=h}
};

```

```

class circle : public rectangle { // türetilmiş sınıf
public:
    void setRadius(int r){Width=Height=r;}
};

```

C++'ın yazılan kodun *yeniden kullanılabilir* olmasını sağlayan mekanizma *kalıtım*dır. Yeniden kullanılabilirlikten, bir sınıfın alınıp bir başka yazılım uygulamasında da (*aynen yada değişikliklerle birlikte*) kullanılabilmesini anlıyoruz. Bu özellik yazılım geliştirme sürecini kısaltırken aynı zamanda yazılımın daha *gülbüz* olmasını sağlayacaktır:

İstemlerin Analizi
Sistem Analizi

Tasarım

Kodlama

Test

Bakım

Türetilmiş Sınıfta Üyelerin Yeniden Tanımlanabilmesi

Bazı durumlarda, temel sınıftaki bir fonksiyonu, türetilmiş sınıfta yeniden tanımlamak gerekebilir:

```
class teacher{    // Base class
public:
    char *Name;
    int Age,numberOfStudents;
    void setName (char *new_name){Name=new_name;}
    void print();
};

void teacher::print()    // Print method of teacher class
{
    cout <<"Name: "<< Name<<" Age: "<< age<< endl;
    cout << "Number of Students: " <<numberOfStudents << endl;
}
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

143

example13.cpp

```
class principal : public teacher{ // Derived class
public:
    char *schoolName;
    int numberOfTeachers;
    void setSchool(char *s_name){schoolName=s_name;}
    void print();    // Print function of principal class
};

void principal::print()    // Print method of principal class
{
    cout <<"Name: "<< Name<<" Age: "<< Age<< endl;
    cout << "Number of Students: " <<numberOfStudents << endl;
    cout <<"Name of the school: "<< schoolName << endl;
}
```

Bu durumda principal sınıfı içinde tanımladığımız yeni print() fonksiyonu temel sınıfta tanımlı print() fonksiyonu üzerine yazacaktır. Eğer temel sınıftaki print() fonksiyonuna erişilmek istenirse :: operatörü kullanılarak teacher::print() yazılır.

C++ ve NESNEYE DAYALI PROGRAMLAMA

144

Örnek

```
class A{
public:
    int ia1,ia2;
    void fa1();
    int fa2(int);
};
```

```
class B: public A{
public:
    float ia1;          // overrides ia1
    float fa1(float);    // overrides fa1
};
```

```
void main(){
    B b;
    int j=b.fa2(1);
    b.ia1=4;           // B::ia1
    b.ia2=3;           // A::ia2 if ia2 is public in A
    float y=b.fa1(3.14); // B::fa1
    b.fa1(); // ERROR fa1 function in B hides the function of A
    b.A::fa1(); // OK
    b.A::ia1=1; // OK
}
```

Erişim Denetimi

Hatırlatma: Bir **sınıf üyesi** (*sınıf içerisindeki*) diğer tüm üyelere erişebilir. O sınıftan bir **nesne** ise sadece **public** ile tanımlı üyelere erişebilir.

Kalıtım mekanizmasında, türetilmiş sınıf üyelerinin, temel sınıf üyelerine erişimi nasıl denetlenebilir?

Kural : türetilmiş sınıf üyeleri temel sınıfın

public ve protected

ile tanımlanmış üyelere erişebilir.

Erişim	Kendi sınıfından erişim	Türetilmiş sınıftan erişim	Dışarıdan erişim
public	evet	evet	evet
protected	evet	evet	hayır
private	evet	hayır	hayır

Genel olarak, üyeleri **private** tanımlamak uygun olacaktır. Böylelikle dışarıdan bir fonksiyonun yanlışlıkla üyenin değerini değiştirmesi olasılığı ortadan kaldırılmış olur. Temel sınıf tasarlanırken olabildiğince **protected** kullanılmasından kaçınılmalıdır. Yeni sınıflar kalıtım yoluyla türetilerek genişletildikçe, üst sınıfların temel sınıf üyelerine erişimi (*karmaşıklık*) önlenmiş olur. Böylelikle daha kararlı ve güvenilir sınıflar gerçekleştirilebilir.

```

class teacher{    // Base class
private:
    char *Name;
protected:
    int Age,numberOfStudents;
public:
    void setName (char *new_name){Name=new_name;}
    void print();
};

class principal : public teacher{ // Derived class
    char *schoolName;
    int numberOfTeachers;
public:
    void setSchool(char *s_name){schoolName=s_name;}
    void print();           // Print function of principal class
    int getAge(){ return Age;} // It works because age is protected
    char * get_name(){ return Name;}
};

```

```
void main()
{
    teacher t1;
    principal p1;
```

```
    t1.numberOfStudents=100;
```

```
    t1.setName("Sema Catir");
```

```
    p1.setSchool("Halide Edip Adivar Lisesi");
```

```
}
```

Public Kalıtım

Kalıtım ile bir sınıf türetilirken, genellikle public takısı kullanılır:

```
class Base
{ };
class Derived : public Base {
```

Bu şekilde türetilen bir sınıfta temel sınıfın üyelik tanımlamaları değişmez. Örneğin temel sınıfın public üyeleri aynı zamanda türetilen sınıfın da public üyeleri olacaktır.

private Kalıtım

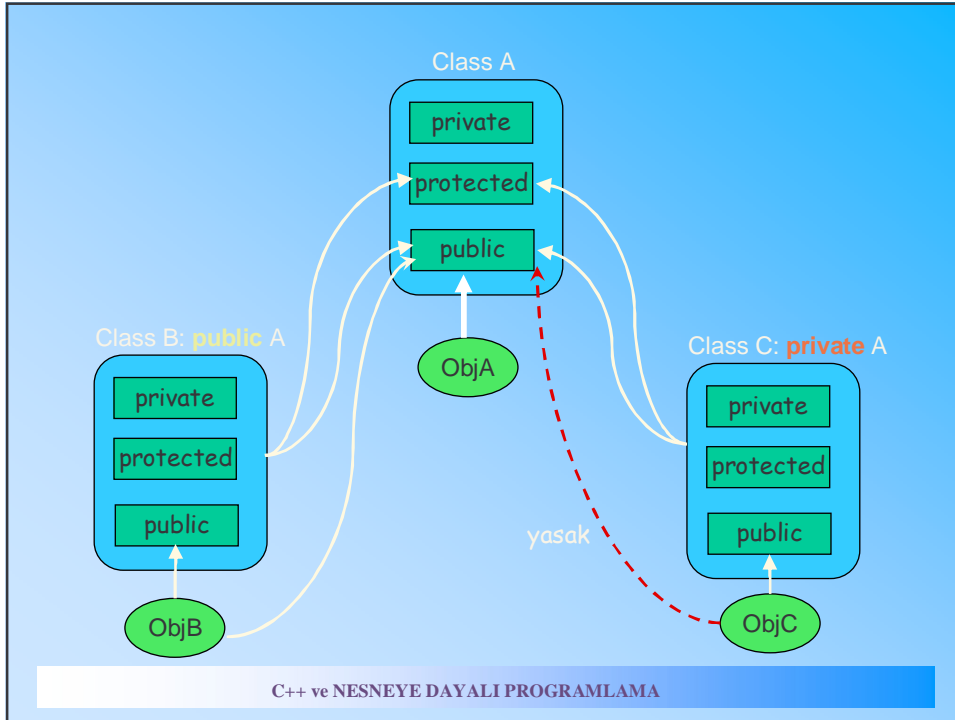
```
class Base
{ };
class Derived : private Base {
```

Buna *private inheritance* denir. Temel sınıfın public üyeleri türetilen sınıfın private üyeleri olur. Bunun sonucu olarak türetilmiş sınıfa ait nesneler temel sınıfın hiçbir elemanına erişemezler. Türetilen sınıfın üyeleri temel sınıfın public and protected tanımlı üyelerine erişebilir.

Türetilmiş Sınıfta Erişimin Yeniden Tanımlanması

Temel sınıfın public tanımlı üyelerine erişim türetilmiş sınıfta yeniden tanımlanabilir.

```
Class Base{
    public:
        void f();
};
class Derived : private Base{ // All members of Base are private now
    int i;
    public:
        Base::f();        // f() is public again
        void fb1();
};
```



Kalıtım ile Aktarılamayan Fonksiyonlar

Temel sınıfta tanımlı bir fonksiyon, eğer işlev yükleme yapılmamış ise, otomatik olarak türetilen sınıf üyelerinin kullanımına aktarılır. Ancak bazı özel fonksiyonlar, kalıtım ile türetilmiş sınıfa aktarılmazlar:

🚦 İşlev yüklenmiş = operatörü

İşlev yüklenmiş atama operatörünün amacını hatırlayınız !

🚦 Kurucu Fonksiyonlar

Temel sınıfın kurucu fonksiyonu türetilmiş sınıfın kurucu fonksiyonu değildir.

🚦 Yokedici Fonksiyonlar

Temel sınıfın yokedici fonksiyonu türetilmiş sınıfın yokedici fonksiyonu değildir.

Kurucu Fonksiyonlar ve Kalıtım

Türetilmiş sınıftan bir nesne yaratıldığında, temel sınıfa ait kurucu fonksiyon türetilmiş sınıfa ait kurucu fonksiyondan önce çağrılır. Temel sınıf üyeleri türetilmiş sınıfın bir alt parçası olduğundan üst parça oluşturulmadan önce alt parçalara ait üyelerin yapılandırılması zorunluluğu vardır.

example15.cpp

```
class teacher{    // turetilmis sinif
    char *Name;
    int Age,numberOfStudents;
public:
    teacher(char *newName){Name=newName;} // temel sinif kurucusu
};

class principal : public teacher{ // turetilmis sinif
    int numberOfTeachers;
public:
    principal(char *, int );    // // turetilmis sinif kurucusu
};
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

155

```
principal::principal(char *new_name,int numOT):teacher(new_name)
{
    numofTeachers=numOT;
}

void main()
{
    principal p1("Sema Catir",20);
}
```

Eğer temel sınıf, parametre alan bir kurucu fonksiyona sahip ise türetilmiş sınıfa ait kurucu fonksiyon, temel sınıf kurucu fonksiyonunu, uygun parametreler ile çağırarak bir kurucuya sahip olmalıdır.

example16.cpp

C++ ve NESNEYE DAYALI PROGRAMLAMA

156

Yokedici Fonksiyonlar ve Kalıtım

Yokedici fonksiyonlar nesnenin erimi dışına çıktığında otomatik olarak çağırılırlar. Kalıtım ile türetilmiş sınıflarda yokedici fonksiyonların çağırılış sırası kurucu fonksiyonların çağırılış sırasının tersi şeklindedir.

Bu durumda ilk olarak türetilmiş sınıfın yokedici fonksiyonu çağırılacaktır.

example17.cpp

C++ ve NESNEYE DAYALI PROGRAMLAMA

157

```
#include <iostream.h>
class B {
public:
    B() { cout << "B constructor" << endl; }
    ~B() { cout << "B destructor" << endl; }
};
class C : public B {
public:
    C() { cout << "C constructor" << endl; }
    ~C() { cout << "C destructor" << endl; }
};
void main(){
    cout << "Start" << endl;
    C ch;    // create a C object
    cout << "End" << endl;
}
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

158

Atama İşlevi ve Kalıtım

Temel sınıfın atama işlevi türetilen sınıfın atama işlevi olamaz.

```
class string {
protected:
    int size;
    char *contents;
public:
    string & operator=(const string &);      // atama islevi
    :    // Other methods
};

string & string::operator=(const string &in_object) {
    size = in_object.size;
    contents = new char[strlen(in_object.contents)+1];
    strcpy(contents, in_object.contents);
    return *this;
}
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

159

example18.cpp

```
class string2 : public string {           // string2 is derived from string
    int size2;
    char *contents2;
public:
    string2 & operator=(const string2 &); // assignment operator for string2
    :    // Other methods
};

/**** Assignment operator for string2 ****/
string2 & string2::operator=(const string2 &in_object) {
    size = in_object.size;                // inherited size
    contents = new char[strlen(in_object.contents)+1]; // inherited contents
    strcpy(contents, in_object.contents);
    size2 = in_object.size2;
    contents2 = new char[strlen(in_object.contents2)+1];
    strcpy(contents2, in_object.contents2);
    return *this;
}
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

160


```
#include <iostream.h>
class A {
private:
    int x;
    float y;
public:
    A(int i, float f) :
        x(i), y(f) // initialize A
    { cout << "Constructor A" << endl; }
    void display() {
        cout << x << ", " << y << "; "; }
};

class B : public A {
private:
    int v;
    float w;
public:
    B(int i1, float f1, int i2, float f2) :
        A(i1, f1), // initialize A
        v(i2), w(f2) // initialize B
    { cout << "Constructor B" << endl; }
    void display(){
        A::display();
        cout << v << ", " << w << "; ";
    }
};
```

Örnek : Sınıf ve Kurucu Zinciri

```
class C : public B {
private:
    int y;
    float z;
public:
    C(int i1,float f1, int i2,float f2,int i3,float f3) :
        B(i1, f1, i2, f2), // initialize B
        y(i3), z(f3) // initialize C
    { cout << "Constructor C" << endl; }
    void display() {
        B::display();
        cout << y << ", " << z;
    }
};
```

```
void main() {
    C c(1, 1.1, 2, 2.2, 3, 3.3);
    cout << "\nData in c = ";
    c.display();
}
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

161

Örnek : Sınıf ve Kurucu Zinciri — Açıklama

C sınıfı B sınıfından ve B sınıfı da A sınıfından türetilmiştir. Her sınıf kendi ve alt sınıflardaki kurucu fonksiyonlarına uygun sayıda parametre almakta ve aktarmaktadır: A sınıfı kurucu fonksiyonu iki, B sınıfı kurucu fonksiyonu dört (ikisi A sınıfı için) ve C sınıfı kurucu fonksiyonu (A ve B sınıfları kurucuları için ikişer parametre) altı parametre almaktadır.

main() fonksiyonunda C sınıfından c adında bir nesne tanımlayıp 6 adet başlangıç değeri verdik. Böylelikle tüm alt sınıflara uygun başlangıç değeri verildi.

```
C(int i1, float f1,int i2, float f2, int i3, float f3) :
    A(i1, f1), // error: can't initialize A
    y(i3), z(f3) // initialize C
{ }
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

162

int ve float gibi basit veri tipine sahip sınıf üyelerine aşağıdaki gibi başlangıç değeri verilebilir :

```
class A{
    int i1,i2;
    A(int new1, int new2): i1(new1),i2(new2) {
        ...
    }
};
```

Ancak bu bir kalıtım uygulaması değildir.

Çoklu Kalıtım

bir sınıfın birden fazla temel sınıftan türetilmesi

```
class Base1{    // Base 1
public:
    int a;
    void fa1();
    char *fa2(int);
};
```

```
class Base2{    // Base 2
public:
    int a;
    char *fa2(int, char);
    int fc();
};
```

```
class Deriv : public Base1, public Base2{
public:
    int a;
    float fa1(float);
    int fb1(int);
};
```

Base1

Base2

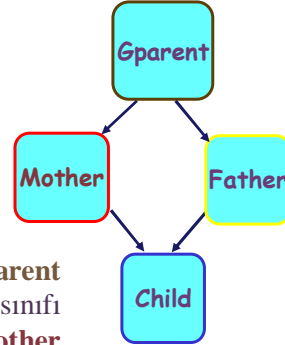
Deriv

```
void main()
{
    Deriv d;
    d.a=4;                //Deriv::a
    float y=d.fa1(3.14); //Deriv::fa1
    int i=d.fc();         // Base2::fc
}
```

char * c=d.fa2(1);
Ataması geçerli değildir.
Kalıtım ile yeniden tanımlanan
fonksiyonlara işlev yüklenemez.
Geçerli kullanım :
char * c=d.Base1::fa2(1);
yada
char * c=d.Base2::fa2(1,"Hello");

Tekrarlı Kalıtım

```
class Gparent
{ };
class Mother : public Gparent
{ };
class Father : public Gparent
{ };
class Child : public Mother, public Father
{ };
```



Hem **Mother** hem de **Father** sınıflarının **Gparent** sınıfından türetildiğine dikkat ediniz. **Child** sınıfı ise çoklu kalıtım ile **Father** ve **Mother** sınıflarından türetilmiştir.

Bu durumda **Gparent** sınıfı hem **Mother** ve hem de **Father** sınıflarında ortak olduğundan **Child** sınıfı iki adet **Gparent** alt sınıfına sahiptir – biri **Father** diğeri **Mother** sınıflarından.

Ayrıca **Gparent** sınıfında aşağıda verildiği gibi bir int tipinde üyeye sahip olsun.



```
class Gparent
{
protected:
    int gdata;
};
class Child : public Mother, public Father
{
public:
    void Cfunc() {
        int temp = gdata; // error: ambiguous
    }
};
```

Derleyici, “Father sınıfından gelen **gdata**’yı mı? yoksa Mother sınıfından gelen **gdata**’yı kullanmalı?” belirsizliği nedeni ile hata verecektir.

Çözüm : Sanal Sınıflar

Bu problem **virtual** anahtar sözcüğü kullanılarak çözülebilir.



```
class Gparent
{ };
class Mother : virtual public Gparent
{ };
class Father : virtual public Gparent
{ };
class Child : public Mother, public Father
{ };
```

virtual anahtar sözcüğü derleyiciye kalıtım ile alt sınıflardan türetilen alt nesnelerden sadece birinin kullanmasını söyler. Ancak bu çözümde burada detaylı olarak duramayacağımız bazı karmaşık durumlarda yeni belirsizlikler getirebilmektedir.

Genel olarak çoklu kalıtmıdan kaçınmalısınız. Ancak C++'da deneyimli iseniz, çoklu kalıtımın gerekli olduğu durumlarda kullanmanız çözümü kolaylaştıracaktır.

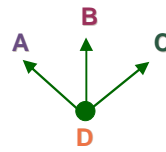
```
class Base
{
public:
    int a,b,c;
};
class Derived : public Base
{
public:
    int b;
};
class Derived2 : public Derived
{
public:
    int c;
};
```



```

class A {
    ...
};
class B {
    ...
};
class C {
    ...
};
class D : public A, public B, private C {
    ...
};

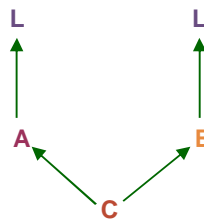
```



```

class L {
    public:
        int next;
};
class A : public L {
    ...
};
class B : public L {
    ...
};
class C : public A, public B {
    void f();
    ...
};

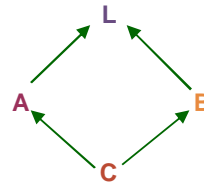
```



```

class L {
public:
    int next;
};
class A : virtual public L {
    ...
};
class B : virtual public L {
    ...
};
class C : public A, public B {
    ...
};

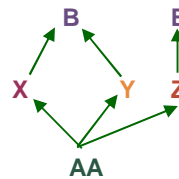
```



```

class B {
    ...
};
class X : virtual public B {
    ...
};
class Y : virtual public B {
    ...
};
class Z : public B {
    ...
};
class AA : public X, public Y, public Z {
    ...
};

```



Sınıf Yapısı

Kalıtım

• Nesne İşaretçileri

Çok Şekillilik

Nesne İşaretçileri

İşaretçiler veri değil verinin yerleşik bulunduğu bellek gözünün adresini taşırlar. İşaretçiler basit tipte değişkenlere işaret edebildikleri gibi bir nesneye de işaret edebilirler. İşaretçiler kullanılmadan önce uygun başlangıç değeri atanmalıdır:

■ new operatörü

İşletim sisteminden uygun miktarda bellek alanı alır. Döndürdüğü değer bu alanın başlangıç adresidir. Eğer işlem başarısız olursa 0 (NULL) döndürür.

Nesne işaretçilerinde new kullanıldığında yukarıdakine ek olarak nesnenin kurucu fonksiyonu çalıştırılır. Böyle nesne yaratılırken başlangıç değerleri atanmış olur.

Nesne İşaretçileri

devam

■ delete operatörü

Belleğin verimli ve etkin kullanımı için, new operatörünün kullanımına karşılık olarak bellek alanı kullanımı bittiğinde işletim sistemine delete operatörü ile geri verilmelidir.

new ile aşağıdaki biçimde bir nesne dizisi için bellek alındığında

```
int * ptr = new int[10];
```

delete ile

```
delete [ ] ptr;
```

şeklinde işletim sistemine geri verilmelidir. İşaretçi önündeki “[]” kullanılmaz ise sadece dizinin ilk elemanı için bellek alanı geri verilir.

C++ ve NESNEYE DAYALI PROGRAMLAMA

175

```
class person          // class of persons
{
    char *name;        // person's name
public:
    person();          //Default Constructor
    void setName(char *); // set the name
    void printName()   // print the name
    {
        cout << "Name is:" << name<<endl;
    }
    ~person()
    { cout << "Destructor" << endl;
      delete name;}
};

person::person()
{
    cout << "Constructor" << endl;
    name = new char;
    name = '\0';
}
```

```
void person::setName(char *n)
{
    delete name;
    name = new char[strlen(n)];
    strcpy(name, n);
}
```

```
void main()
{
    person* persPtr = new person[3];
    persPtr->setName("Person1");
    (persPtr+1)->setName("Person2");
    (persPtr+2)->setName("Person3");
    persPtr->printName();
    (persPtr+1)->printName();
    (persPtr+2)->printName();
    delete [ ] persPtr;
} // end main()
```

example22.cpp

C++ ve NESNEYE DAYALI PROGRAMLAMA

176

Nesne Bağlantılı Listeleri

Bir sınıf kendi tipinden bir nesneye işaretçi içerebilir. Bu işaretçi kullanılarak bir nesne zinciri (bağlantılı liste) kurulabilir.

```
class teacher{
    friend class teacher_list;
    char *name;
    int age,numOfStudents;
    teacher * next;    // Pointer to next object of teacher
public:
    teacher(char *, int, int); // Constructor
    void print();
    char *getName(){return name;}
    ~teacher()          // Destructor
    {
        cout<<" Destructor of teacher" << endl;
        delete name;
    }
};
```



example23.cpp

```
/* linked list for teachers */
class teacher_list{
    teacher *head;
public:
    teacher_list(){head=0;}
    char append(char *,int,int);
    char del(char *);
    void print();
};
```

İşaretçiler ve Kalıtım

Eğer bir sınıf temel bir sınıftan türetilmiş ise, türetilmiş sınıftan bir işaretçiye herhangi bir tip dönüşümü gerekmez. Temel sınıftan bir işaretçi atanabilir. Temel sınıfın işaretçisi türetilmiş sınıfa işaretçi olabilir. Tersi bir dönüşüm, tip dönüşümü gerektirir.

Örneğin, teacher nesnesine bir işaretçi teacher ve principal nesnelere işaret edebilir. principal ile teacher arasında "is a" ilişkisi vardır : principal is a teacher. Ancak tersi her zaman doğru değildir : her teacher bir principal olmayabilir.



```
class Base{
};
class Derived : public Base {
};

Derived d,*dp;
Base *bp=&d;    // implicit conversion
dp=bp;          // error Base is not Derived
dp = (Derived *)bp; // explicit conversion
```

Eğer bir sınıf kalıtım ile private olarak temel sınıftan türetilirse, bu durumda tip dönüşümü yapılamaz. Çünkü temel sınıfın public üyeleri temel sınıfa ait işaretçiler tarafından erişilebilir. Ancak türetilmiş sınıftan nesneler yada işaretçiler temel sınıf üyelerine erişemezler.



```
class Base{
    int m1;
public:
    int m2;
};
class Derived : private Base { // m2 is not a public member of Derived
};

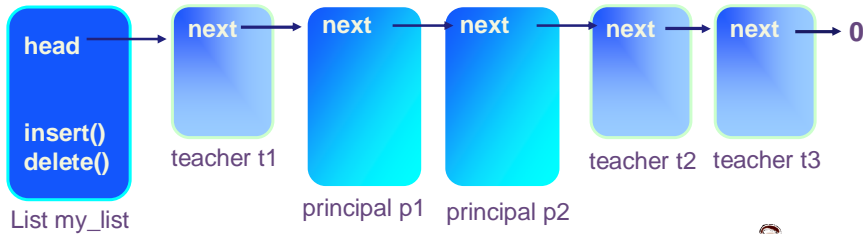
Derived d;
d.m2=5;           // error m2 is private member of Derived
base *bp=&d;      // error private base
bp->m2=5;          // ok
bp = (base*)&d;    // ok: explicit conversion
bp->m2=5;          // ok
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

179

İşaretçiler ve kalıtım kullanılarak, heterojen bağlantılı listeler oluşturulabilir. Temel sınıfa işaret eden nesnelerden oluşan liste, bu temel sınıftan türetilmiş tüm sınıflara ait nesneler içerebilir. Heterojen listeleri daha sonra çok şekillilik konusunda tekrar inceleyeceğiz.

Örnek: öğretmenler ve müdürler listesi

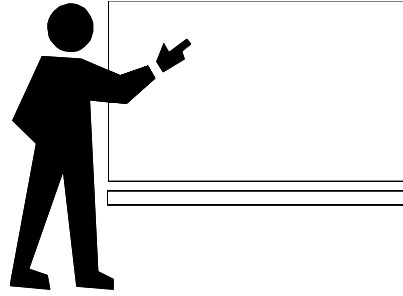


example24.cpp

C++ ve NESNEYE DAYALI PROGRAMLAMA

180

UYGULAMA



C++ ve NESNEYE DAYALI PROGRAMLAMA

181

Sınıf Yapısı

Kalıtım

3

Çok Şekillilik

Nesneye dayalı programlamanın üç temel kavramı :

1. Sınıflar,
2. Kalıtım,
3. **Çok Şekillilik** (C++'da *sanal fonksiyonlar* ile sağlanır)

C++ ve NESNEYE DAYALI PROGRAMLAMA

182

ÇOK ŞEKİLLİLİK (=POLYMORPHISM)

Gerçek hayattaki nesneler farklı sınıflardan olsalar da, yada farklı davranışları gerçekleştirseler de, bazı ortak işlevlere sahip olabilmektedirler. Örnek olarak **kare**, **daire**, **üçgen** sınıflarından nesneleri ele alalım:

Tüm bu nesnelere “Alan Hesapla” mesajını göndermiş olalım. Farklı tipte nesneler (**kare**, **daire**, **üçgen**) farklı alan hesabına sahiptir. Ancak farklı tipte nesnelere farklı mesaj göndermeye gerek yoktur. Bu işlem için tek bir tür mesaj (**Area()**) tüm nesnelerde için çalışmalıdır. Çünkü her bir tip nesne kendi sınıfından nesnelerin alanını nasıl hesaplayacağını bilmektedir:

kare→**Area()** ;

daire→**Area()**;

üçgen→**Area()**;

C++ ve NESNEYE DAYALI PROGRAMLAMA

183

ÇOK ŞEKİLLİLİK

devam

Area() işlevi farklı tipte nesnelerde farklı biçimler aldığından **çok şekillik** gösteren bir methodur.

Çok şekillilik, genellikle birbirleri ile kalıtımla ilişkili sınıflar arasında gerçekleşir. C++’da çok şekilliliğin anlamı, bir üye fonksiyona yapılan çağrının, farklı nesnelerde, nesnenin tipine bağlı olarak farklı fonksiyonların çağrılmasına neden olmasıdır.


Bu biraz, fonksiyona işlev yüklemeyi çağrıştırmaktadır. Ancak çok şekillilik daha güçlü ve farklı bir mekanizma sunmaktadır. Önemli fark, hangi fonksiyonun çağrılacağına ne zaman karar verildiğinde ortaya çıkmaktadır.

Fonksiyon yüklemede bu karar, derleyici tarafından derleme aşamasında verilirken, çok şekillikte bu karar yürütme zamanında verilmektedir.

C++ ve NESNEYE DAYALI PROGRAMLAMA

184

Normal Sınıf Üyelerine Çok Şekillilik Mekanizması Kullanılmadan İşaretçiler ile Erişim




↓

```

class Square {    // Temel sınıf
protected:
    double edge;
public:
    Square(double e):edge(e){ } // temel sınıf kurucusu
    double Area() { return( edge * edge ); }
};

class Cube : public Square { // Türetilmiş sınıf
public:
    Cube(double e):Square(e){ } // Türetilmiş sınıf kurucusu
    double Area() { return( 6.0 * edge * edge ); }
};
      
```



C++ ve NESNEYE DAYALI PROGRAMLAMA

185

```

void main(){
    Square S(2.0)           ;
    Cube  C(8.0)           ;
    Square *ptr            ;
    char   c                ;
    cout << "Square or Cube"; cin >> c ;
    if (c=='s') ptr=&S      ;
    else  ptr=&C            ;
    ptr->Area();           // which Area ???
}
      
```

Cube sınıfı Square sınıfından türetilmiştir. Her iki sınıfta Area() üye mesajını içermektedir. main() fonksiyonunda, Square ve Cube sınıflarından birer nesne ve Square sınıfına bir işaretçi tanımlanmıştır. Ardından türetilmiş sınıftan nesnenin adresi temel sınıfa işaret eden işaretçiye atanmıştır:

ptr = &C;

Bu geçerli bir atamadır

square.cpp

C++ ve NESNEYE DAYALI PROGRAMLAMA

186

Aşağıdaki satır yürütüldüğünde

```
ptr→Area();
```

Square::Area() fonksiyonu mu?

yoksa

Cube::Area() fonksiyonu mu?

çağırılır.

İşaretçiler ile Erişilen Virtual Üye Fonksiyonları

Şimdi programda tek bir değişiklik yapalım: temel sınıftaki **Area()** fonksiyonunun önüne **virtual** anahtar sözcüğünü koyalım.

```
class Square { // Temel sınıf
protected:
    double edge;
public:
    Square(double e):edge(e){ } // temel sınıf kurucusu
    virtual double Area(){ return( edge * edge ) ; }
};
class Cube : public Square { // Türetilmiş sınıf
public:
    Cube(double e):Square(e){ } // Türetilmiş sınıf kurucusu
    double Area(){ return( 6.0 * edge * edge ) ; }
};
```

```

void main(){
    Square S(2.0)                ;
    Cube C(8.0)                  ;
    Square *ptr                  ;
    char c                       ;

    cout << "Square or Cube"; cin >> c ;
    if (c=='s') ptr=&S            ;
        else ptr=&C              ;
    ptr->Area();
}

```

Şimdi ptr'nin içeriğine göre farklı fonksiyonlar çağırılacaktır. Fonksiyonlar ptr'nin tipine göre değil, içeriğine göre çağırılmaktadır. Bu şekilde çok şekillilik sağlanır. virtual anahtar sözcüğü, Area() fonksiyonunun çok şekilli olmasını sağlamaktadır.

Peki derleyici hangi fonksiyonu çağıracağını derleme zamanında nasıl biliyor? virtual anahtar sözcüğünü kullanmadığımız ilk örnekte bir sorun yoktu: ptr->Area(); her zaman temel sınıftaki Area() fonksiyonu çalıştırılır. Ama ikinci örnekte, derleyici ptr işaretçisinin hangi tipten bir sınıfın nesnesine işaret ettiğini bilmektedir. ptr yürütme zamanında Square sınıfından yada Cube sınıfından bir nesneye işaret edebilir. Bu durumda hangi Area() çalıştırılır? Derleyici derleme esnasında bu kararı veremeyeceğinden, yürütme zamanında bu kararı verecek şekilde kodu düzenler.

Yürütme zamanında, fonksiyon çağırısı oluştuğunda, derleyicinin yerleştirdiği bir kod, ptr nesne işaretçisinin hangi tipten bir nesneye işaret ettiğini algılar ve ilgili fonksiyonu çağırır: Square::Area() yada Cube::Area().

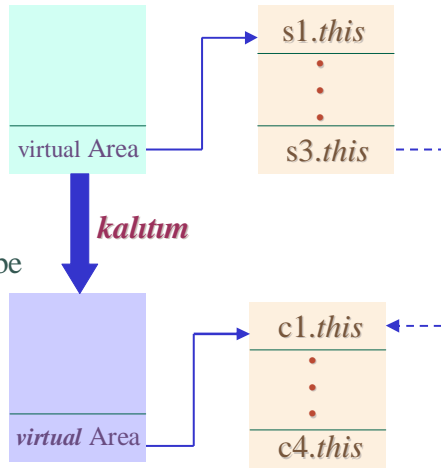
Buna late binding yada dynamic binding adı verilmektedir. Late binding bir miktar ek işlem yükü getirmektedir (yaklaşık %10 gibi). Ama bunun karşılığında yazdığımız programa büyük bir esneklik ve yetenek kazandırmaktadır.

Virtual tanımlaması içermeyen bir sınıfa ait nesne sadece üye alanaları bellekte yerleşik bulunur. İlgili nesne için bir üye fonksiyona çağrı yapıldığında, derleyici nesnenin adresini fonksiyona parametre olarak aktarır. Bu adresin **this** değişkeninde saklı bulunduğunu hatırlayınız. Derleyici, fonksiyonunun formal parametrelerine ek olarak (programcıya gizli bir biçimde) **this** işaretçisini ekler. Bu parametre her ilgili fonksiyon çağrısında derleyici tarafından aktarılacaktır; **this** işaretçisi nesnenin üyeleri ile fonksiyonları arasındaki yegane bağlantıyı oluşturmaktadır.

virtual tanımlı fonksiyonlar ise biraz daha karmaşık bir davranış göstermektedir. Türetilmiş sınıfta tanımlanan her virtual fonksiyon için derleyici, **Virtual Table** adı verilen bir dizi (tablo) oluşturur. **Square** ve **Cube** sınıflarının her biri **Virtual Table** dizisine sahiptir. Bu dizilerde, o sınıftaki her bir virtual fonksiyon için bir kayıt bulunur (tutulur).

```
Square s1(1),s2(4),s3(3);
Cube c1(2),c2(7) ;
Cube c3(8),c4(8) ;
Square *p ;
...
p = &c2 ;
...
p->Area() ;
...
p = &s3 ;
...
p->Area() ;
```

Square



Cube

Bu örnekte, **Square** yada **Cube** sınıfından bir nesnenin virtual tanımlı fonksiyonuna bir çağrı yapıldığında, uygun üye fonksiyonunun çağırılması için gerekli işlemleri derleme aşamasında derleyici yerine, derleyicinin ürettiği bir kod yürütme zamanında gerçekleştirmektedir. Üretilen bu kod, nesnenin virtual tablosunu tarayarak, uygun üye fonksiyona erişimi sağlamaktadır.

Bunu Nesneler ile Denemeyin !

Sanal fonksiyon mekanizması sadece nesne işaretçileri ile kullanımda çalışır.

```
void main()
{
    Square S(4);
    Cube C(8);
    S.Area();
    C.Area();
}
```

Uyarı

```
class Square {    // Temel sınıf
protected:
    double edge;
public:
    Square(double e):edge(e){ } // temel sınıf kurucusu
    virtual double Area(){ return( edge * edge ) ; }
};
class Cube : public Square { // Türetilmiş sınıf
public:
    Cube(double e):Square(e){ } // Türetilmiş sınıf kurucusu
    double Area(){ return( 6.0 * Square::Area() ) ; }
};
```

*Burada **Square::Area()** virtual değil*



Nesne Bağlantılı Listesi ve Çok Şekillilik

Sanal fonksiyonların en çok kullanım alanı bulunduğu uygulama bağlantılı nesne liste yapılarıdır:

```
class Square {    // Temel sınıf
protected:
    double edge;
public:
    Square(double e):edge(e){ } // temel sınıf kurucusu
    virtual double Area(){ return( edge * edge ) ; }
    Sqaure *next ;
};
class Cube : public Square { // Türetilmiş sınıf
public:
    Cube(double e):Square(e){ } // Türetilmiş sınıf kurucusu
    double Area(){ return( 6.0 * edge * edge ) ; }
};
```

```

void main(){
    Cube c1(50);
    Square s1(40);
    Cube c2(23);
    Square s2(78);
    Square *listPtr;    // Pointer of the linked list
    /** Construction of the list ***/
    listPtr=&c1;
    c1.next=&s1;
    s1.next=&c2;
    c2.next=&s2;
    s2.next=0L;
    /** Printing all elements of the list ***/
    while (listPtr){
        cout << listPtr->Area() << endl ;
        listPtr=listPtr->next;
    }
}

```

C++ ve NESNEYE DAYALI PROGRAMLAMA example27.cpp

197

Abstract Classes

To write polymorphic functions we need to have derived classes. But sometimes we don't need to create any base class objects, but only derived class objects. The base class exists only as a starting point for deriving other classes. This kind of base classes we can call an **abstract class**, which means that no actual objects will be created from it. Abstract classes arise in many situations. A factory can make a sports car or a truck or an ambulance, but it can't make a generic vehicle. The factory must know the details about what *kind* of vehicle to make before it can actually make one. Similarly, you'll see sparrows, wrens, and robins flying around, but you won't see any generic birds. Actually, a class is an abstract class only in the eyes of humans. The compiler is ignorant of our decision to make it an abstract class.

Pure Virtual Functions

It would be nice if, having decided to create an abstract base class, I could instruct the compiler to actively *prevent* any class user from ever making an object of that class. This would give me more freedom in designing the base class because I wouldn't need to plan for actual objects of the class, but only for data and functions that would be used by derived classes. There is a way to tell the compiler that a class is abstract: You define at least one **pure virtual function** in the class.

A pure virtual function is a virtual function with no body. The body of the virtual function in the base class is removed, and the notation =0 is added to the function declaration.

C++ ve NESNEYE DAYALI PROGRAMLAMA

198

Örnek

```
class generic_shape{      // Abstract base class
protected:
    int x,y;
public:
    generic_shape(int x_in,int y_in){ x=x_in; y=y_in;} // Constructor
    virtual void draw()=0;    // pure virtual function
};

class Line:public generic_shape{    // Line class
protected:
    int x2,y2;          // End coordinates of line
public:
    Line(int x_in,int y_in,int x2_in,int y2_in):generic_shape(x_in,y_in)
    {
        x2=x2_in;
        y2=y2_in;
    }
    void draw(){ line(x,y,x2,y2); }    // virtual draw function
};
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

199

```
class Rectangle:public Line{      // Rectangle class
public:
    Rectangle(int x_in,int y_in,int x2_in,int y2_in):Line(x_in,y_in,x2_in,y2_in){}
    void draw(){ rectangle(x,y,x2,y2); }    // virtual draw
};

class Circle:public generic_shape{    // Circle class
protected:
    int radius;
public:
    Circle(int x_cen,int y_cen,int r):generic_shape(x_cen,y_cen)
    {
        radius=r;
    }
    void draw() { circle(x,y, radius); }    // virtual draw
};

/* A function to draw different shapes */
void show(generic_shape &shape)
{
    // Which draw function will be called?
    shape.draw();    // It 's unknown at compile-time
}
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

200

```
void main()
{
    int gdriver = DETECT, gmode, errorcode;
    initgraph(&gdriver, &gmode, "\\tc\\bgi"); //To graphics mode
    Line Line1(1,1,100,250);
    Circle Circle1(100,100,20);
    Rectangle Rectangle1(30,50,250,140);
    Circle Circle2(300,170,50);
    show(Circle1);
    getch();
    show(Line1);
    getch();
    show(Circle2);
    getch();
    show(Rectangle1);
    getch();
    closegraph();
}
```

Sanal Fonksiyonlar ve Kurucu Fonksiyonlar

Kurucu Fonksiyonlar Sanal olabilir mi?

Hayır.

Bir nesne yaratıldığında, derleyici bu nesnenin hangi sınıftan olduğunu bilmektedir. Bu nedenle, sanal kurucu fonksiyonlara ihtiyaç yoktur.

Sanal Yokedici Fonksiyonlar

```
class Base {
    public:
        ~Base() { cout << "\nBase destructor"; }
};
class Derived : public Base {
    public:
        ~Derv() { cout << "\nDerived destructor"; }
};
void main(){
    Base* pb = new Derived;
    delete pb;
    cout << endl << "Program terminates." << endl ;
}
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

203

```
class Base {
    public:
        virtual ~Base() { cout << "\nBase destructor"; }
};
class Derived : public Base {
    public:
        ~Derv() { cout << "\nDerived destructor"; }
};
void main(){
    Base* pb = new Derived;
    delete pb;
    cout << endl << "Program terminates." << endl ;
}
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

204

Sınıf Yapısı

Kalıtım

Çok Şekillilik

4 ≡

Templates

Parametrik Çok Şekillilik Nedir ?

Sınıflardaki fonksiyonların gövdeleri incelendiğinde, çoğu zaman yapılan işlemler, üzerinde işlem yapılan verinin tipinden bağımsızdır. Bu durumda fonksiyonun gövdesi, verinin tipi cinsinden, parametrik olarak ifade edilebilir:

```
int abs(int n) {  
    return (n<0) ? -n : n;  
}  
  
long abs(long n) {  
    return (n<0) ? -n : n;  
}  
  
float abs(float n) {  
    return (n<0) ? -n : n;  
}
```

- **C**

Her tip için farklı adlarda fonksiyonlar.

örnek mutlak değer fonksiyonları:

abs(), **fabs()**, **fabsl()**, **labs()**, **cabs()**, ...

- **C++**

Fonksiyonlara işlev yükleme bir çözüm olabilir mi?

İşlev yüklenen fonksiyonların gövdeleri değişmiyor !

Gövdeler tekrar ediliyor \Rightarrow Hata !

- **Çözüm**

Tipi parametre kabul eden bir yapı : Template

```
#include <iostream.h>
template <class T>
T abs(T n) {
    return (n < 0) ? -n : n;
}
void main()
{
    int int1 = 5;
    int int2 = -6;
    long lon1 = 70000L;
    long lon2 = -80000L;
    double dub1 = 9.95;
    double dub2 = -10.15;
    // calls instantiate functions
    cout << "abs(" << int1 << ")=" << abs(int1) << endl;    // abs(int)
    cout << "abs(" << int2 << ")=" << abs(int2) << endl;    // abs(int)
    cout << "abs(" << lon1 << ")=" << abs(lon1) << endl;    // abs(long)
    cout << "abs(" << lon2 << ")=" << abs(lon2) << endl;    // abs(long)
    cout << "abs(" << dub1 << ")=" << abs(dub1) << endl;    // abs(double)
    cout << "abs(" << dub2 << ")=" << abs(dub2) << endl;    // abs(double)
}
```



```

template <class T>
    void printArray(T *array, const int size){
        for(int i=0; i < size; i++)
            cout << array[i] << " ";
        cout << endl ;
    }

```

```

main() {
    int    a[3]={1,2,3} ;
    double b[5]={1.1,2.2,3.3,4.4,5.5} ;
    char   c[7]={'a', 'b', 'c', 'd', 'e', 'f', 'g'} ;

    printArray(a,3)    ;
    printArray(b,5)    ;
    printArray(c,7)    ;

    return 0 ;
}

```

```

void printArray(int *array,cont int size){
    for(int i=0;i < size;i++)
        cout << array[i] << " " ;
    cout << endl ;
}
void printArray(char *array,cont int size){
    for(int i=0;i < size;i++)
        cout << array[i] << "" ;
    cout << endl ;
}

```

template'in İşleyişi

Gerçekte derleyici template ile verilmiş fonksiyon gövdesi için herhangi bir kod üretmez. Çünkü template ile bazı verilerin tipi parametrik olarak ifade edilmiştir. Verinin tipi ancak bu fonksiyona ilişkin bir çağrı olduğunda ortaya çıkacaktır. Derleyici her farklı tip için yeni bir fonksiyon oluşturacaktır. template yeni fonksiyonun verinin tipine bağlı olarak nasıl oluşturulacağını tanımlamaktadır.

```

int int1 = 5;
cout << "abs(" << int << ")=" << abs(int1);

```

programı ister **template** yapısı ile oluşturalım ister de **template** yapısı olmaksızın oluşturalım, programın bellekte kaplayacağı alan değişmeyecektir. Değişen, kaynak kodun boyu olacaktır. **template** yapısı kullanılarak oluşturulan programın kaynak kodu, daha anlaşılır ve hata denetimi daha yüksek olacaktır. Çünkü **template** yapısı kullanıldığında değişiklik sadece tek bir fonksiyon gövdesinde yapılacaktır.

template Parametresi bir Nesne Olabilir

```
class ComplexT{                                /* A class to define complex numbers */
    float re,im;
public:
    : // other member functions
    bool operator>(const ComplexT&) const ; // header of operator> function
};

/* The Body of the function for operator + */
bool ComplexT::operator>(const ComplexT& z) const
{
    float f1 = re * re + im * im;
    float f2 = z.re * z.re + z.im * z.im;
    if (f1 > f2) return true;
    else      return false;
}
```

```

// template function
template <class type>
const type & MAX(const type &v1, const type & v2)
{
    if (v1 > v2) return v1;
    else      return v2;
}

void main()
{
    int i1=5, i2= -3;
    char c1='D', c2='N';
    float f1=3.05, f2=12.47;
    ComplexT z1(1.4,0.6), z2(4.6,-3.8);
    cout << MAX(i1,i2) << endl;
    cout << MAX(c1,c2) << endl;
    cout << MAX(f1,f2) << endl;
    cout << MAX(z1,z2) << endl;
}

```

Çoklu template Parametrelili Argümanlar

```

// function returns index number of item, or -1 if not found template
template <class atype>
int find(const atype *array, atype value, int size) {
    for(int j=0; j<size; j++)
        if(array[j]==value) return j;
    return -1;
}

char chrArr[] = {'a', 'c', 'f', 's', 'u', 'z'}; // array
char ch = 'f'; // value to find
int intArr[] = {1, 3, 5, 9, 11, 13};
int in = 6;
double dubArr[] = {1.0, 3.0, 5.0, 9.0, 11.0, 13.0};
double db = 4.0;

```

```

void main()
{
    cout << "\n 'f' in chrArray: index=" << find(chrArr, ch, 6);
    cout << "\n 6 in intArray: index=" << find(intArr, in, 6);
    cout << "\n 4 in dubArray: index=" << find(dubArr, db, 6);
}

```

template İmzaları

```

template <class T>
void swap(T& x, T& y) {
    T temp ;
    temp = x ;
    x = y ;
    y = temp ;
}
char str1[100], str2[100] ;
int i,j ;
complex c1,c2;
swap( i , j ) ;
swap( c1 , c2 ) ;
swap( str1[50] , str2[50] ) ;
swap( i , str[25] ) ;
swap( str1 , str2 ) ;

```

Çoklu template Parametrelili Yapılar

Template parametre sayısı birden fazla olabilir:

```
template <class atype, class btype>
btype find(const atype* array, atype value, btype size) {
    for(btype j=0; j<size; j++) // note use of btype
        if(array[j]==value) return j;
    return (btype)-1;
}
```

Bu durumda, derleyici sadece farklı dizi tipleri için değil aynı zamanda aranan elemanın farklı tipte olması durumunda da farklı bir kod üretecektir:

```
short int result, si=100;
int inval=5;
result = find(intArr, inval, si);
```

Sınıf Template Yapısı

```
class Stack {
    int st[MAX]; // array of ints
    int top; // index number of top of stack
public:
    Stack(); // constructor
    void push(int var); // takes int as argument
    int pop(); // returns int value
};

class LongStack {
    long st[MAX]; // array of longs
    int top; // index number of top of stack
public:
    LongStack(); // constructor
    void push(long var); // takes long as argument
    long pop(); // returns long value
};
```

```

template <class Type>
class Stack{
    enum {MAX=100};
    Type st[MAX];           // stack: array of any type
    int top;                // number of top of stack
public:
    Stack(){top = 0;}       // constructor
    void push(Type );       // put number on stack
    Type pop();             // take number off stack
};
template<class Type>
void Stack<Type>::push(Type var) // put number on stack
{
    if(top > MAX-1)         // if stack full,
        throw "Stack is full!"; // throw exception
    st[top++] = var;
}

```

C++ ve NESNEYE DAYALI PROGRAMLAMA

221

```

template<class Type>
Type Stack<Type>::pop() {    // take number off stack
    if(top <= 0)             // if stack empty,
        throw "Stack is empty!"; // throw exception
    return st[--top];
}

```

```

void main()
{
    // s1 is object of class Stack<float>
    Stack<float> s1;
    // push 2 floats, pop 2 floats
    try{
        s1.push(1111.1);
        s1.push(2222.2);
        cout << "1: " << s1.pop() << endl;
        cout << "2: " << s1.pop() << endl;
    }
    // exception handler
    catch(const char * msg) {
        cout << msg << endl;
    }
}

```

```

// s2 is object of class Stack<long>
Stack<long> s2;
// push 2 longs, pop 2 longs
try{
    s2.push(123123123L);
    s2.push(234234234L);
    cout << "1: " << s2.pop() << endl;
    cout << "2: " << s2.pop() << endl;
}
// exception handler
catch(const char * msg) {
    cout << msg << endl;
}
// End of program

```

C++ ve NESNEYE DAYALI PROGRAMLAMA

222

Sınıf Yapısı

Kalıtım

Çok Şekillilik

Templates

5 ≡

STL Kütüphanesi ve “Generic Programming”
(Standard Template Library)

Standard Template Library

Nesneye dayalı programlamada, verinin birincil öneme sahip programlama birimi olduğunu belirtmiştik. Veri, fiziksel yada soyut bir çok büyüklüğü modelleyebilir. Bu model oldukça basit yada karmaşık olabilir. Her nasıl olursa olsun, veri mutlaka bellekte saklanmaktadır ve veriye benzer biçimlerde erişilmektedir. C++, oldukça karmaşık veri tiplerini ve yapılarını oluşturmamıza olanak sağlayan mekanizmalara sahiptir. Genel olarak, programların, bu veri yapılarına belirli bazı biçimlerde eriştiğini biliyoruz:

`array`, `list`, `stack`, `queue`, `vector`, `map`, ...

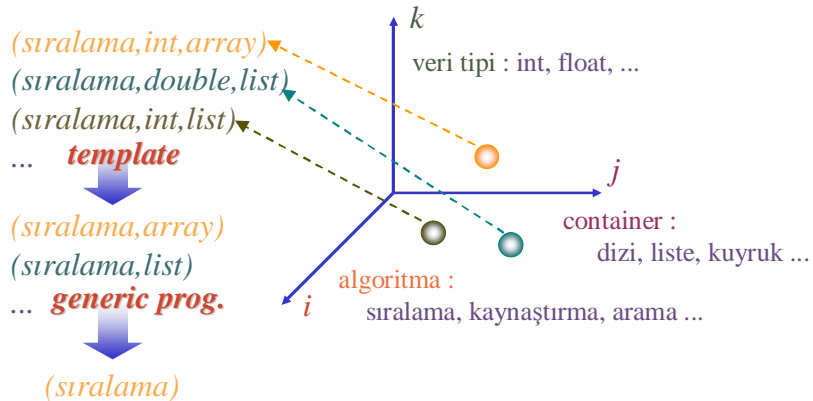
STL kütüphanesi verinin bellekteki organizasyonuna, erişimine ve işlenmesine yönelik çeşitli yöntemler sunmaktadır. Bu bölümde bu yöntemleri inceleyeceğiz.

Standard Template Library (STL) Hewlett Packard'ın Palo Alto (California)'daki laboratuvarlarında Alexander Stepanov ve Meng Lee tarafından geliştirilmiştir.

1970'lerin sonlarında Alexander Stepanov bir kısım algoritmaların veri yapısının nasıl depolandıklarından bağımsız olduklarını gözlemledi. Örneğin, sıralama algoritmalarında sıralanacak sayıların bir dizide mi? yoksa bir listede mi? bulunduğunun bir önemi yoktur. Değişen sadece bir sonraki elemana nasıl erişildiği ile ilgilidir. Stepanov bu ve benzeri algoritmaları inceleyerek, algoritmaları veri yapısından bağımsız olarak performanstan ödün vermeksizin soyutlamayı başarmıştır. Bu fikrini 1985'de Generic ADA dilinde gerçekleştirmiştir. Ancak o dönemde henüz C++'da bir önceki bölümde incelediğimiz Template yapısı bulunmadığı için bu fikrini C++'da ancak 1992 yılında gerçekleştirebilmiştir.

Generic Programming

Bir yazılım ürününün bileşenlerini üç boyutlu uzayda bir nokta olarak düşünebiliriz :



STL Bileşenleri

STL üç temel bileşenden oluşmaktadır:

- Algoritma,
- Kap (= Container): nesneleri depolamak ve yönetmekten sorumlu nesne,
 - Lineer Kaplar : Vector, Deque, List
 - Asosyatif Kaplar : Set, Map, Multi-set, Multi-map
- Yineleyici (=Iterator): algoritmanın farklı tipte kaplarla çalışmasını sağlayacak şekilde erişimin soyutlar.

C++'da sabit boyutlu dizi tanımlamak yürütme zamanında belleğin ya kötü kullanılmasına ya da dizi boyunun yetersiz kalmasına neden olmaktadır.

STL kütüphanesindeki **vector** kabı bu sorunları gidermektedir.

STL kütüphanesindeki **list** kabı, bağlantılı liste yapısıdır.

deque (*Double-Ended QUEUE*) kabı, yığın ve kuyruk yapılarının birleşimi olarak düşünülebilir. deque kabı her iki uçtan veri eklemeye ve silmeye olanak sağlamaktadır.

Vector	Relocating, expandable array	Quick random access (by index number). Slow to insert or erase in the middle. Quick to insert or erase at end.
List	Doubly linked list	Quick to insert or delete at any location. Quick access to both ends. Slow random access.
Deque	Like vector, but can be accessed at either end	Quick random access (using index number). Slow to insert or erase in the middle. Quick to insert or erase (push and pop) at either the beginning or the end.

Lineer Kaplar : Vector, List, Deque

```
vector<float> v;
cout << v.capacity() << v.size() ;
v.insert(v.end(),3) ;
cout << v.capacity() << v.size() ;
v.insert (v.begin(), 2, 5);
```

v = (3)

v = (3,5,5)

```
vector<int> w (4,9);
w.insert(w.end(), v.begin(), v.end() );
w.swap(v) ;
```

w = (9,9,9,9)

w = (9,9,9,9,3,5,5)

v = (9,9,9,9,3,5,5)

w=(3,5,5)

```
w.erase(w.begin());
w.erase(w.begin(),w.end() );
cout << w.empty() ? "Empty" : "not Empty"
```

w = (3,5)

Empty

```
#define __USE_STL
```

```
// STL include files
```

```
#include "vector.h"
```

```
#include "list.h"
```

```
vector<float> v;
v.insert(v.end(),3) ;
v.insert(v.begin(),5) ;
cout << v.front() << endl;
cout << v.back() ;
v.pop_back();
cout << v.back() ;
```

v = (3)

v = (3,5)

5

3

5

İşlem	Yürütülen İşlem
a.size()	a.end() – a.begin()
a.max_size()	
a.empty()	a.size() == 0

İşlem	Dönüş Değeri	Yürütülen İşlem	Uygulanabildiği Kaplar
a.front()	T&	*a.begin()	vector, list, deque
a.back()	T&	*a.end()	vector, list, deque
a.push_front(x)	void	a.insert(a.begin(),x)	list,deque
a.push_back(x)	void	a.insert(a.begin(),x)	vector, list,deque
a.pop_front()	void	a.erase(a.begin())	list,deque
a.pop_back()	void	a.erase(--a.end())	list,deque
a[n]	T&	*(a.begin()+n)	vector,deque

Asosyatif Kaplar : Set, Multiset, Map, Multimap

Set sıralı küme oluşturmak için kullanılır.

```
#include <iostream>
#include <set>
#include <string>
using namespace std;
void main(){
    string names[] = {"Katie", "Robert", "Mary", "Amanda", "Marie"};
    set<string> nameSet(names, names+5); // initialize set to array
    set<string>::const_iterator iter; // iterator to set
    nameSet.insert("Jack");           // insert some more names
    nameSet.insert("Larry");
    nameSet.insert("Robert");         // no effect; already in set
    nameSet.insert("Barry");
    nameSet.erase("Mary");           // erase a name
```

```

cout << "\nSize=" << nameSet.size() << endl;
iter = nameSet.begin();          // display members of set
while( iter != nameSet.end() )
    cout << *iter++ << '\n';
string searchName;               // get name from user
cout << "\nEnter name to search for: ";
cin >> searchName;              // find matching name in set
iter = nameSet.find(searchName);
if( iter == nameSet.end() )
    cout << "The name " << searchName << " is NOT in the set.";
else
    cout << "The name " << *iter << " IS in the set.";
}

```

```

// set2.cpp set
void main() {
    set<string> city;
    set<string>::iterator iter;
    city.insert("Trabzon");    // insert city names
    city.insert("Adana");
    city.insert("Edirne");
    city.insert("Bursa");
    city.insert("Istanbul");
    city.insert("Rize");
    city.insert("Antalya");
    city.insert("İzmir");
    city.insert("Hatay");
    city.insert("Ankara");
    city.insert("Zonguldak");
}

```

```

iter = city.begin();          // display set
while( iter != city.end() )
    cout << *iter++ << endl;

string lower, upper;          // display entries in range
cout << "\nEnter range (example A Azz): ";
cin >> lower >> upper;
iter = city.lower_bound(lower);
while( iter != city.upper_bound(upper) )
    cout << *iter++ << endl;
}

```

C++ ve NESNEYE DAYALI PROGRAMLAMA

235

```

void main(){
    map<string,int> city_num;
    city_num["Trabzon"]=61;
    ...
    string city_name;
    cout << "\nEnter a city: ";
    cin >> city_name;
    if (city_num.end()== city_num.find(city_name))
        cout << city_name << " is not in the database" << endl;
    else
        cout << "Number of " << city_name << ": " << city_num[city_name];
}

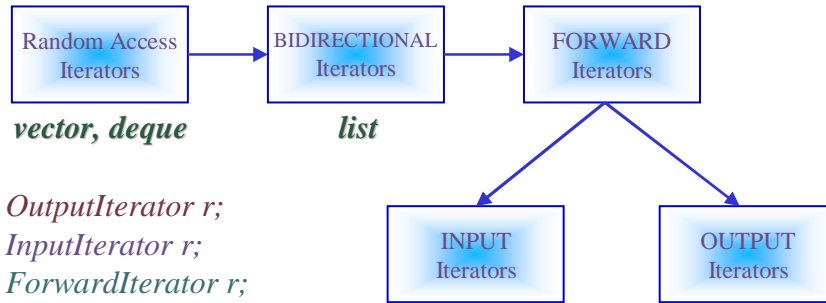
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

236

Iterators

Iterators : Genelleştirilmiş İşaretçi



OutputIterator r;
InputIterator r;
ForwardIterator r;
BidirectionalIterator r;
RandomIterator r;

Output Iterators

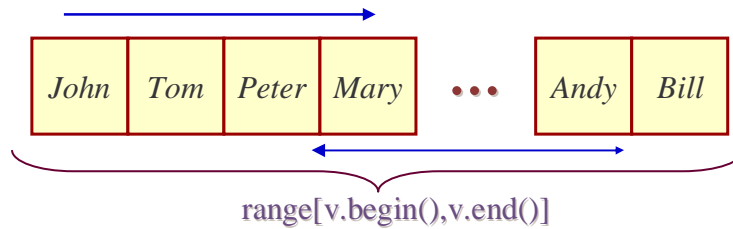
① *OutputIterator a ;*
 ...
 *a=t ;
 t = *a ; *Hata*

② *OutputIterator r ;*
 ...
 *r=0 ;
 *r=1 ; *Hata*

③ *OutputIterator r ;*
 ...
 r++ ;
 r++ ; *Hata*

④ *OutputIterator i,j ;*
 ...
 i=j ;
 *i++=a ; *Hata*
 *j=b ;

Forward and Bidirectional Iterators



```
list<int> l (1,1) ;
l.push_back(2) ; // list l : 1 2
list<int>::iterator first=l.begin() ;
list<int>::iterator last=l.end() ;
while( last != first){
    -- last ;
    cout << *last << " " ;
}
```

```
template<class ForwardIterator, class T>
ForwardIterator find_linear(ForwardIterator first,
                           ForwardIterator last, T& value){
    while( first != last) if( *first++ == value) return first;
    else return last ;
}

vector<int> v(3,1) ;
v.push_back(7); // vector : 1 1 1 7
vector<int>::iterator i=find_linear(v.begin(), v.end(),7) ;
if(i != v.end() ) cout << *i ;
else cout << "not found!" ;
```


Bubble Sort

```
template<class Compare>
void bubble_sort(BidirectionalIterator first,
                 BidirectionalIterator last, Compare comp){
    BidirectionalIterator left = first , right = first ;
    right ++ ;
    while( first != last){
        while( right != last ){
            if( comp(*right,*left) )
                iter_swap(left,right) ;
            right++ ;
            left++;
        }
        last -- ;
        left = first ; right = first ;
    }
}
```

```
list<int> l ;
bubble_sort(l.begin(),l.end(),less<int>()) ;
bubble_sort(l.begin(),l.end(),greater<int>()) ;
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

241

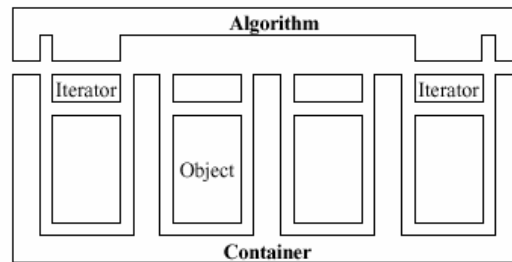
Random Access Iterators

```
vector<int> v(1,1) ;
v.push_back(2) ; v.push_back(3) ; v.push_back(4) ; // v : 1 2 3 4
vector<int>::iterator i=v.begin() ;
vector<int>::iterator j=i+2;
cout << *j << " " ;
i += 3 ; cout << *i << " " ;
j = i - 1 ; cout << *j << " " ;
j -= 2 ; cout << *j << " " ;
cout << v[1] << endl ;
(j<i) ? cout << "j < i" : cout << "not j < i" ; cout << endl ;
(j>i) ? cout << "j > i" : cout << "not j > i" ; cout << endl ;
(j>=i) && (j<=i)? cout << "j and i equal" : cout << "j and i not equal > i" ;
cout << endl ;
i = j ;
j= v.begin();
i = v.end ;
cout << "iterator distance end – begin : " << (i-j) ;
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

242

Generic Algoritma Tasarımı



```
const int * binary_search(const int * array, int n, int x){
    const int *lo = array, *hi = array + n , *mid ;
    while( lo != hi ) {
        mid = lo + (hi-lo)/2 ;
        if( x == *mid ) return mid ;
        if( x < *mid ) hi = mid ;
        else lo = mid + 1 ;
    }
    return 0 ;
}
```

```

template<class T>
const T * binary_search(const T * array, int n, T& x){
    const T *lo = array, *hi = array + n , *mid ;
    while( lo != hi ) {
        mid = lo + (hi-lo)/2 ;
        if( x == *mid ) return mid ;
        if( x < *mid ) hi = mid ;
        else lo = mid + 1 ;
    }
    return 0 ;
}

```

```

template<class T>
const T * binary_search(T * first,T * last, T& x){
    const T *lo = first, *hi = last , *mid ;
    while( lo != hi ) {
        mid = lo + (hi-lo)/2 ;
        if( x == *mid ) return mid ;
        if( x < *mid ) hi = mid ;
        else lo = mid + 1 ;
    }
    return last ;
}

```

```

template<class RandomAccessIterator,class T>
const T * binary_search(RandomAccessIterator first,
                        RandomAccessIterator last, T& value){
    RandomAccessIterator not_found = last, mid ;
    while( lo != hi ) {
        mid = first + (last-first)/2 ;
        if( x == *mid ) return mid ;
        if( x < *mid ) last = mid ;
        else first = mid + 1 ;
    }
    return not_found ;
}

```

Sınıf Yapısı

Kalıtım

Çok Şekillilik

Templates

STL Kütüphanesi ve “Generic Programming”

6 ≡ Stream Kütüphanesi

STREAMS

A *stream* is a general name given to a flow of data in an input/output situation. For this reason, streams in C++ are often called *iostreams*. An *iostream* can be represented by an object of a particular class. For example, you've already seen numerous examples of the `cin` and `cout` stream objects used for input and output.

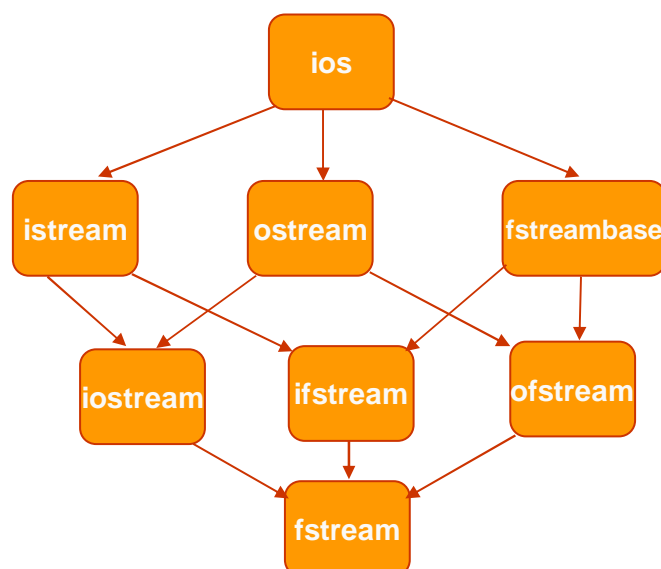
Advantages of Streams

Old-fashioned C programmers may wonder what advantages there are to using the stream classes for I/O instead of traditional C functions such as `printf()` and `scanf()` and—for files—`fprintf()`, `fscanf()`, and so on.

One reason is that the stream classes are less prone to errors. If you've ever used a `%d` formatting character when you should have used a `%f` in `printf()`, you'll appreciate this. There are no such formatting characters in streams, because each object already knows how to display itself. This removes a major source of program bugs.

Second, you can overload existing operators and functions, such as the insertion (`<<`) and extraction (`>>`) operators, to work with classes you create. This makes your classes work in the same way as the built-in types, which again makes programming easier and more error free (not to mention more aesthetically satisfying).

The Stream Class Hierarchy



The `ios` class is the base class for the `iostream` hierarchy. It contains many constants and member functions common to input and output operations of all kinds. The `ios` class also contains a pointer to the `streambuf` class, which contains the actual memory buffer into which data is read or written and the low-level routines for handling this data.

The `istream` and `ostream` classes are derived from `ios` and are dedicated to input and output, respectively. The `istream` class contains such member functions as `get()`, `getline()`, `read()`, and the extraction (`>>`) operators, whereas `ostream` contains `put()` and `write()` and the insertion (`<<`) operators.

The `iostream` class is derived from both `istream` and `ostream` by multiple inheritance. Classes derived from the `iostream` class can be used with devices, such as disk files, that may be opened for both input and output at the same time.

The `ifstream` class is used for creating input file objects and the `ofstream` class is used for creating output file objects. To create a read/write file the `fstream` class should be used.

The ios Class

The `ios` class is the grand daddy of all the stream classes and contains the majority of the features you need to operate C++ streams. The three most important features are the formatting flags, the error-status bits, and the file operation mode. We'll look at formatting flags and error-status bits now.

Formatting Flags

Formatting flags are a set of enum definitions in `ios`. They act as on/off switches that specify choices for various aspects of input and output format and operation.

<code>skipws</code>	Skip (ignore) whitespace on input.
<code>left</code>	Left adjust output.
<code>right</code>	Right adjust output.
<code>dec</code>	Convert to decimal.
<code>oct</code>	Convert to octal.
<code>hex</code>	Convert to hexadecimal.
<code>showbase</code>	Use base indicator on output (0 for octal, 0x for hex).
<code>showpoint</code>	Show decimal point on output.
<code>uppercase</code>	Use uppercase X, E, and hex output letters ABCDEF.
<code>showpos</code>	Display '+' before positive integers.
<code>scientific</code>	Use exponential format on floating-point output [9.1234E2].
<code>fixed</code>	Use fixed format on floating-point output [912.34].
<code>unitbuf</code>	Flush all streams after insertion.

There are several ways to set the formatting flags, and different flags can be set in different ways. Because they are members of the `ios` class, flags must usually be preceded by the name `ios` and the scope-resolution operator (e.g., `ios::skipws`). All the flags can be set using the `setf()` and `unsetf()` `ios` member functions. For example,

```
cout.setf(ios::left);    // left justify output text
cout >> "This text is left-justified";
cout.unsetf(ios::left); // return to default (right justified)
```

Many formatting flags can be set using manipulators, so let's look at them now.

Manipulators

Manipulators are formatting instructions inserted directly into a stream. You've seen examples before, such as the manipulator `endl`, which sends a new line to the stream and flushes it:

```
cout << "To each his own." << endl;
```

There is also used the `setiosflags()` manipulator:

```
cout << setiosflags(ios::fixed) // use fixed decimal point
      << setiosflags(ios::showpoint) //always show decimal point
      << var;
```

No-argument ios Manipulators

<code>ws</code>	Turn on whitespace skipping on input.
<code>dec</code>	Convert to decimal.
<code>oct</code>	Convert to octal.
<code>hex</code>	Convert to hexadecimal.
<code>endl</code>	Insert new line and flush the output stream.
<code>ends</code>	Insert null character to terminate an output string.
<code>flush</code>	Flush the output stream.
<code>lock</code>	Lock file handle.
<code>unlock</code>	Unlock file handle.

You insert these manipulators directly into the stream. For example, to output `var` in hexadecimal format, you can say

```
cout << hex << var;
```

ios manipulators with arguments

setw()	field width (int)	Set field width for output.
setfill()	fill character (int)	Set fill character for output (default is a space).
setprecision()	precision (int)	Set precision (number of digits displayed). setiosflags()
formatting	flags (long)	Set specified flags.
resetiosflags()	formatting flags (long)	Clear specified flags.

Manipulators that take arguments affect only the next item in the stream. For example, if you use `setw` to set the width of the field in which one number is displayed, you'll need to use it again for the next number.

Functions

The `ios` class contains a number of functions that you can use to set the formatting flags and perform other tasks. most of these functions are shown below:

<code>ch = fill();</code>	Return the fill character (fills unused part of field; default is space).
<code>fill(ch);</code>	Set the fill character.
<code>p = precision()</code>	Get the precision (number of digits displayed for floating point).
<code>precision(p);</code>	Set the precision.
<code>w = width();</code>	Get the current field width (in characters).
<code>width(w);</code>	Set the current field width.
<code>setf(flags);</code>	Set specified formatting flags (e.g., <code>ios::left</code>).
<code>unsetf(flags);</code>	Unset specified formatting flags.

C++ ve NESNEYE DAYALI PROGRAMLAMA

255

These functions are called for specific stream objects using the normal dot operator. For example, to set the field width to 14, you can say

```
cout.width(14);
```

Similarly, the following statement sets the fill character to an asterisk (as for check printing):

```
cout.fill('*');
```

You can use several functions to manipulate the `ios` formatting flags directly. For example, to set left justification, use

```
cout.setf(ios::left);
```

To restore right justification, use

```
cout.unsetf(ios::left);
```

The istream Class

The `istream` class, which is derived from `ios`, performs input-specific activities.

istream functions:

<code>>></code>	Formatted extraction for all basic (and overloaded) types.
<code>get(ch);</code>	Extract one character into <code>ch</code> .
<code>get(str)</code>	Extract characters into array <code>str</code> , until <code>'\0'</code> .
<code>get(str, MAX)</code>	Extract up to <code>MAX</code> characters into array.
<code>get(str, DELIM)</code>	Extract characters into array <code>str</code> until specified delimiter (typically <code>'\n'</code>).
	Leave delimiting char in stream.

C++ ve NESNEYE DAYALI PROGRAMLAMA

256

istream functions:

<code>get(str, MAX, DELIM)</code>	Extract characters into array str until MAX characters or the DELIM character. Leave delimiting char in stream.
<code>getline(str, MAX, DELIM)</code>	Extract characters into array str until MAX characters or the DELIM character. Extract delimiting character.
<code>putback(ch)</code>	Insert last character read back into input stream.
<code>ignore(MAX, DELIM)</code>	Extract and discard up to MAX characters until (and including) the specified delimiter (typically '\n').
<code>peek(ch)</code>	Read one character, leave it in stream.
<code>count = gcount()</code>	Return number of characters read by a (immediately preceding) call to <code>get()</code> , <code>getline()</code> , or <code>read()</code> .
<code>read(str, MAX)</code>	For files. Extract up to MAX characters into str until EOF.
<code>seekg(position)</code>	Sets distance (in bytes) of file pointer from start of file.
<code>seekg(position, seek_dir)</code>	Sets distance (in bytes) of file pointer from specified place in file: <code>seek_dir</code> can be <code>ios::beg</code> , <code>ios::cur</code> , <code>ios::end</code> .
<code>position = tellg(pos)</code>	Return position (in bytes) of file pointer from start of file.

The ostream Class

The ostream class handles output or insertion activities.

ostream functions:

<code><<</code>	Formatted insertion for all basic (and overloaded) types.
<code>put(ch)</code>	Insert character ch into stream.
<code>flush()</code>	Flush buffer contents and insert new line.
<code>write(str, SIZE)</code>	Insert SIZE characters from array str into file.
<code>seekp(position)</code>	Sets distance in bytes of file pointer from start of file.
<code>seekp(position, seek_dir)</code>	Set distance in bytes of file pointer from specified place in file. <code>seek_dir</code> can be <code>ios::beg</code> , <code>ios::cur</code> , or <code>ios::end</code> .
<code>position = tellp()</code>	Return position of file pointer, in bytes.

The istream and the _withassign Classes

The istream class, which is derived from both istream and ostream, acts only as a base class from which other classes, specifically istream_withassign, can be derived. It has no functions of its own (except constructors and destructors). Classes derived from istream can perform both input and output.

There are three _withassign classes:

`istream_withassign`, derived from `istream`
`ostream_withassign`, derived from `ostream`
`iostream_withassign`, derived from `iostream`

These _withassign classes are much like those they're derived from except they include overloaded assignment operators so their objects can be copied.

Predefined Stream Objects

Objects Name	Class	Used for
cin	istream_withassign	Keyboard input
cout	ostream_withassign	Normal screen output
cerr	ostream_withassign	Error output
clog	ostream_withassign	Log output

The cerr object is often used for error messages and program diagnostics. Output sent to cerr is displayed immediately, rather than being buffered, as output sent to cout is. Also, output to cerr cannot be redirected. For these reasons, you have a better chance of seeing a final output message from cerr if your program dies prematurely. Another object, clog, is similar to cerr in that it is not redirected, but its output is buffered, whereas cerr's is not.

Stream Errors

What happens if a user enters the string "nine" instead of the integer 9, or pushes ENTER without entering anything? What happens if there's a hardware failure? We'll explore such problems in this session. Many of the techniques you'll see here are applicable to file I/O as well.

Error-Status Bits

The stream error-status bits (error byte) are an ios member that report errors that occurred in an input or output operation.

goodbit	No errors (no bits set, value = 0).
eofbit	Reached end of file.
failbit	Operation failed (user error, premature EOF).
badbit	Invalid operation (no associated streambuf).
hardfail	Unrecoverable error.

Various ios functions can be used to read (and even set) these error bits.

int = eof();	Returns true if EOF bit set.
int = fail();	Returns true if fail bit or bad bit or hard-fail bit set.
int = bad();	Returns true if bad bit or hard-fail bit set.
int = good();	Returns true if everything OK; no bits set.
clear(int=0);	With no argument, clears all error bits; otherwise sets specified bits, as in clear(ios::failbit).

```

#include <iostream.h>
void main()
{
    int i;
    char ok=0;
    while(!ok)                // cycle until input OK
    {
        cout << "\nEnter an integer: ";
        cin >> i;
        if( cin.good() )      // if no errors
            ok=1;
        else
        {
            cin.clear();        // clear the error bits
            cout << "Incorrect input";
            cin.ignore(20, '\n'); // remove newline
        }
    }
    cout << "integer is " << i; // error-free integer
}

```

See Example: inp.cpp

No-Input Input

Whitespace characters, such as TAB, ENTER , and '\n', are normally ignored (skipped) when inputting numbers. This can have some undesirable side effects. For example, users, prompted to enter a number, may simply press the key without typing any digits. Pressing ENTER causes the cursor to drop down to the next line while the stream continues to wait for the number. What's wrong with the cursor dropping to the next line? First, inexperienced users, seeing no acknowledgment when they press , may assume the computer is broken. Second, pressing repeatedly normally causes the cursor to drop lower and lower until the entire screen begins to scroll upward. Thus it's important to be able to tell the input stream *not* to ignore whitespace. This is done by clearing the skipws flag:

```

cout << "\nEnter an integer: ";
cin.unsetf(ios::skipws); // don't ignore whitespace
cin >> i;
if( cin.good() )
{
    // no error
}
// error

```

Now if the user types without any digits, failbit will be set and an error will be generated. The program can then tell the user what to do or reposition the cursor so the screen does not scroll.

Disk File I/O with Streams

Disk files require a different set of classes than files used with the keyboard and screen. These are `ifstream` for input, `fstream` for input and output, and `ofstream` for output. Objects of these classes can be associated with disk files and you can use their member functions to read and write to the files.

The `ifstream`, `ofstream`, and `fstream` classes are declared in the `FSTREAM.H` file. This file also includes the `IOSTREAM.H` header file, so there is no need to include it explicitly; `FSTREAM.H` takes care of all stream I/O.

```
#include <fstream.h>           // for file I/O
void main()
{
    char ch = 'x';              // character
    int j = 77;                 // integer
    double d = 6.02;           // floating point
    char str1[] = "Kafka";      // strings
    char str2[] = "Proust";     // (no embedded spaces)
    ofstream outfile("fdata.txt"); // create ofstream object
    outfile << ch               // insert (write) data
        << j << ' '           // needs space between numbers
        << d
        << str1 << ' '        // needs space between strings
        << str2;
}
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

263

Here the program defines an object called `outfile` to be a member of the `ofstream` class. At the same time, it initializes the object to the file name `FDATA.TXT`. This initialization sets aside various resources for the file, and accesses or *opens* the file of that name on the disk. If the file doesn't exist, it is created. If it does exist, it is truncated and the new data replaces the old. The `outfile` object acts much as `cout` did in previous programs, so the insertion operator (`<<`) is used to output variables of any basic type to the file. This works because the insertion operator is appropriately overloaded in `ostream`, from which `ofstream` is derived.

When the program terminates, the `outfile` object goes out of scope. This calls its destructor, which closes the file, so you don't need to close the file explicitly.

You must separate numbers (such as 77 and 6.02) with nonnumeric characters. Because numbers are stored as a sequence of characters rather than as a fixed-length field, this is the only way the extraction operator will know, when the data is read back from the file, where one number stops and the next one begins. Second, strings must be separated with whitespace for the same reason. This implies that strings cannot contain embedded blanks. In this example, I use the space character (" ") for both kinds of delimiters. Characters need no delimiters, because they have a fixed length.

C++ ve NESNEYE DAYALI PROGRAMLAMA

264

Reading Data

Any program can read the file generated by previous program by using an ifstream object that is initialized to the name of the file. The file is automatically opened when the object is created. The program can then read from it using the extraction (>>) operator.

```
// reads formatted output from a file, using >>
#include <fstream.h>
const int MAX = 80;
void main()
{
    char ch;           // empty variables
    int j;
    double d;
    char str1[MAX];
    char str2[MAX];
    ifstream infile("fdata.txt"); // create ifstream object
    infile >> ch >> j >> d >> str1 >> str2; // extract (read) data from it
    cout << ch << endl           // display the data
        << j << endl
        << d << endl
        << str1 << endl
        << str2 << endl;
}
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

265

Detecting End-of-File

Objects derived from ios contain error-status bits that can be checked to determine the results of operations. When you read a file little by little, you will eventually encounter an end-of-file condition. The EOF is a signal sent to the program from the hardware when there is no more data to read. The following construction can be used to check for this:

```
while( !infile.eof() ) // until eof encountered
```

However, checking specifically for an eofbit means that I won't detect the other error bits, such as the failbit and badbit, which may also occur, although more rarely. To do this, I could change the loop condition:

```
while( infile.good() ) // until any error encountered
```

But even more simply, I can test the stream directly

```
while( infile ) // until any error encountered
```

Any stream object, such as infile, has a value that can be tested for the usual error conditions, including EOF. If any such condition is true, the object returns a zero value. If everything is going well, the object returns a nonzero value. This value is actually a pointer, but the "address" returned has no significance except to be tested for a zero or nonzero value.

C++ ve NESNEYE DAYALI PROGRAMLAMA

266

Binary I/O

You can write a few numbers to disk using formatted I/O, but if you're storing a large amount of numerical data, it's more efficient to use binary I/O in which numbers are stored as they are in the computer's RAM memory rather than as strings of characters. In binary I/O an integer is always stored in 2 bytes, whereas its text version might be 12345, requiring 5 bytes. Similarly, a float is always stored in 4 bytes, whereas its formatted version might be 6.02314e13, requiring 10 bytes.

The next example shows how an array of integers is written to disk and then read back into memory using binary format. I use two new functions: `write()`, a member of `ofstream`, and `read()`, a member of `ifstream`. These functions think about data in terms of bytes (type `char`). They don't care how the data is formatted, they simply transfer a buffer full of bytes from and to a disk file. The parameters to `write()` and `read()` are the address of the data buffer and its length. The address must be cast to type `char`, and the length is the length in bytes (characters), *not* the number of data items in the buffer.

Example

```
// binary input and output with integers
#include <fstream.h>           // for file streams
const int MAX = 100;         // number of ints
int buff[MAX];               // buffer for integers
void main()
{
    int j;
    for(j=0; j<MAX; j++)      // fill buffer with data
        buff[j] = j; // (0, 1, 2, ...)
    ofstream os("edata.dat", ios::binary); // create output stream
    os.write( (char*)buff, MAX*sizeof(int) ); // write to it
    os.close(); // must close it
    for(j=0; j<MAX; j++)      // erase buffer
        buff[j] = 0;
    ifstream is("edata.dat", ios::binary); // create input stream
    is.read( (char*)buff, MAX*sizeof(int) ); // read from it
    for(j=0; j<MAX; j++) // check data
        if( buff[j] != j ) cerr << "\nData is incorrect";
        else cout << "\nData is correct";
}
```

Writing an Object to Disk

When writing an object, you generally want to use binary mode. This writes the same bit configuration to disk that was stored in memory and ensures that numerical data contained in objects is handled properly.

```
// saves person object to disk
#include <fstream.h>           // for file streams

class person                  // class of persons
{
protected:
    char name[40];            // person's name
    int age;                  // person's age
public:
    void getData(void)        // get person's data
    {
        cout << "Enter name: "; cin >> name;
        cout << "Enter age: "; cin >> age;
    }
};
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

269

```
void main(void)
{
    person pers;              // create a person
    pers.getData();           // get data for person
    ofstream outfile("PERSON.DAT", ios::binary); // create ofstream object
    outfile.write( (char*)&pers, sizeof(pers) ); // write to it
}
```

Reading an Object from Disk

```
// reads person object from disk
#include <fstream.h>          // for file streams
class person                 // class of persons
{
protected:
    char name[40];           // person's name
    int age;                 // person's age
public:
    void showData(void)      // display person's data
    {
        cout << "\n  Name: " << name;
        cout << "\n  Age: " << age;
    }
};
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

270

```

void main(void)
{
    person pers;           // create person variable
    ifstream infile("PERSON.DAT", ios::binary); // create stream
    infile.read( (char*)&pers, sizeof(pers) ); // read stream
    pers.showData();        // display person
}

```

To work correctly, programs that read and write objects to files, must be working on the same class of objects. Objects of class person in these programs are exactly 42 bytes long, with the first 40 occupied by a string representing the person's name and the last 2 containing an int representing the person's age.

Notice, however, that although the person classes in both programs have the same data, they may have different member functions. The first includes the single function getData(), whereas the second has only showData(). It doesn't matter what member functions you use, because members functions are not written to disk along with the object's data. The data must have the same format, but inconsistencies in the member functions have no effect. This is true only in simple classes that don't use virtual functions.

I/O with Multiple Objects

```

// reads and writes several objects to disk
#include <fstream.h>           // for file streams
class person                  // class of persons
{
    protected:
        char name[40];        // person's name
        int age;              // person's age
    public:
        void getData()        // get person's data
        {
            cout << "\n Enter name: "; cin >> name;
            cout << " Enter age: "; cin >> age;
        }
        void showData()       // display person's data
        {
            cout << "\n Name: " << name;
            cout << "\n Age: " << age;
        }
};

```



```

void main()
{
    char ch;
    person pers;           // create person object
    fstream file;          // create input/output file
    file.open("PERSON.DAT", ios::app | ios::out | ios::in | ios::binary ); // open for append
    do{                     // data from user to file
        cout << "\nEnter person's data:";
        pers.getData();     // get one person's data
        file.write( (char*)&pers, sizeof(pers) ); // write to file
        cout << "Enter another person (y/n)? ";
        cin >> ch;
    } while(ch!='y');       // quit on 'n'
    file.seekg(0);          // reset to start of file
    file.read( (char*)&pers, sizeof(pers) ); // read first person
    while( !file.eof() )    // quit on EOF
    {
        cout << "\nPerson:"; // display person
        pers.showData();
        file.read( (char*)&pers, sizeof(pers) ); // read another
        // person
    }
}

```

See Example: objfile.cpp

Reacting to Errors

The next program shows how errors are most conveniently handled. All disk operations are checked after they are performed. If an error has occurred, a message is printed and the program terminates. We will use the technique, discussed earlier, of checking the return value from the object itself to determine its error status. The program opens an output stream object, writes an entire array of integers to it with a single call to write(), and closes the object. Then it opens an input stream object and reads the array of integers with a call to read().

```

// handles errors during input and output
#include <fstream.h> // for file streams
#include <process.h> // for exit()
const int MAX = 1000;
int buff[MAX];
void main()
{
    int j;
    for(j=0; j<MAX; j++) buff[ j ] = j; // fill buffer with data
    ofstream os; // create output stream
    os.open("edata.dat", ios::trunc | ios::binary); // open it
    if(!os) { cerr << "\nCould not open output file"; exit(1); }
    cout << "\nWriting..."; // write buffer to it
    os.write( (char*)buff, MAX*sizeof(int) );
    if(!os) { cerr << "\nCould not write to file"; exit(1); }
    os.close(); // must close it
}

```

```

for(j=0; j<MAX; j++) buff[ j ] = 0; // clear buffer
ifstream is; // create input stream
is.open("edata.dat", ios::binary);
if(!is) { cerr << "\nCould not open input file"; exit(1); }
cout << "\nReading...";
is.read( (char*)buff, MAX*sizeof(int) ); // read file
if(!is) { cerr << "\nCould not read from file"; exit(1); }
for(j=0; j<MAX; j++) // check data
    if( buff[j] != j ) { cerr << "\nData is incorrect"; exit(1); }
cout << "\nData is correct";
}

```

Analyzing Errors

In the previous example, we determined whether an error occurred in an I/O operation by examining the return value of the entire stream object.

```

if(!is)
    // error occurred

```

However, it's also possible, using the ios error-status bits, to find out more specific information about a file I/O error.

```

// checks for errors opening file
#include <fstream.h> // for file functions

void main()
{
    ifstream file;
    file.open("GROUP.DAT", ios::nocreate);

    if( !file )
        cout << endl << "Can't open GROUP.DAT";
    else
        cout << endl << "File opened successfully.";
    cout << endl << "file = " << file;
    cout << endl << "Error state = " << file.rdstate();
    cout << endl << "good() = " << file.good();
    cout << endl << "eof() = " << file.eof();
    cout << endl << "fail() = " << file.fail();
    cout << endl << "bad() = " << file.bad();
    file.close();
}

```

This program first checks the value of the object file. If its value is zero, the file probably could not be opened because it didn't exist. Here's the output of the program when that's the case:

```
Can't open GROUP.DAT
```

```
file = 0x1c730000
```

```
Error state = 4
```

```
good() = 0
```

```
eof() = 0
```

```
fail() = 4
```

```
bad() = 4
```

The error state returned by `rdstate()` is 4. This is the bit that indicates the file doesn't exist; it's set to 1. The other bits are all set to 0. The `good()` function returns 1 (true) only when no bits are set, so it returns 0 (false). I'm not at EOF, so `eof()` returns 0. The `fail()` and `bad()` functions return nonzero because an error occurred.

In a serious program, some or all of these functions should be used after every I/O operation to ensure that things have gone as expected.

File Pointers

Each file object has associated with it two integer values called the *get pointer* and the *put pointer*. These are also called the *current get position* and the *current put position*, or—if it's clear which one is meant—simply the *current position*. These values specify the byte number in the file where writing or reading will take place.

There are times when you must take control of the file pointers yourself so that you can read from or write to an arbitrary location in the file. The `seekg()` and `tellg()` functions allow you to set and examine the get pointer, and the `seekp()` and `tellp()` functions perform the same actions on the put pointer.

```
// seeks particular person in file
#include <fstream.h> // for file streams
class person // class of persons
{
protected:
    char name[40]; // person's name
    int age; // person's age
public:
    void showData() // display person's data
    {
        cout << "\n Name: " << name; cout << "\n Age: " << age;
    }
};
```

```

void main()
{
    person pers; // create person object
    ifstream infile; // create input file
    infile.open("PERSON.DAT", ios::binary); // open file
    infile.seekg(0, ios::end); // go to 0 bytes from end
    int endposition = infile.tellg(); // find where we are
    int n = endposition / sizeof(person); // number of persons
    cout << endl << "There are " << n << " persons in file";
    cout << endl << "Enter person number: "; cin >> n;
    int position = (n-1) * sizeof(person); // number times size
    infile.seekg(position); // bytes from begin
    infile.read( (char*)&pers, sizeof(pers) ); // read one person
    pers.showData(); // display the person
}

```

Here's the output from the program, assuming that the PERSON.DAT file contains 3 persons:

```

There are 3 persons in file
Enter person number: 2
    Name: Rainier
    Age: 21

```

File I/O Using Member Functions

So far, we've let the main() function handle the details of file I/O. This is nice for demonstrations, but in real object-oriented programs, it's natural to include file I/O operations as member functions of the class.

In the next example, we will add member functions, diskOut() and diskIn() to the person class. These functions allow a person object to write itself to disk and read itself back in.

Simplifying assumptions: First, all objects of the class will be stored in the same file, called PERSON.DAT. Second, new objects are always appended to the end of the file. An argument to the diskIn() function allows me to read the data for any person in the file. To prevent attempts to read data beyond the end of the file, I include a static member function, diskCount(), that returns the number of persons stored in the file.

```

// person objects do disk I/O
#include <fstream.h> // for file streams
class person // class of persons
{
protected:
    char name[40]; // person's name
    int age; // person's age
public:
    void getData() // get person's data
    { cout << "\n Enter name: "; cin >> name; cout << " Enter age: "; cin >> age; }
    void showData() // display person's data
    { cout << "\n Name: " << name; cout << "\n Age: " << age; }
    void diskIn(int ); // read from file
    void diskOut(); // write to file
    static int diskCount(); // return number of persons in file
};

void person::diskIn(int pn) // read person number pn from file
{
    ifstream infile; // make stream
    infile.open("PERSON.DAT", ios::binary); // open it
    infile.seekg( pn*sizeof(person) ); // move file ptr
    infile.read( (char*)this, sizeof(*this) ); // read one person
}

```

C++ ve NESNEYE DAYALI PROGRAMLAMA

281

```

void person::diskOut() // write person to end of file
{
    ofstream outfile; // make stream
    outfile.open("PERSON.DAT", ios::app | ios::binary); // open it
    outfile.write( (char*)this, sizeof(*this) ); // write to it
}

int person::diskCount() // return number of persons in file
{
    ifstream infile;
    infile.open("PERSON.DAT", ios::binary);
    infile.seekg(0, ios::end); // go to 0 bytes from end
    return infile.tellg() / sizeof(person); // calculate number of persons
}

```

C++ ve NESNEYE DAYALI PROGRAMLAMA

282

```

void main(void)
{
    person p;           // make an empty person
    char ch;
    do                  // save persons to disk
    {
        cout << "\nEnter data for person:";
        p.getData();    // get data
        p.diskOut();    // write to disk
        cout << "Do another (y/n)? ";
        cin >> ch;
    }
    while(ch=='y');     // until user enters 'n'

    int n = person::diskCount(); // how many persons in file?
    cout << "\nThere are " << n << " persons in file";
    for(int j=0; j<n; j++) // for each one,
    {
        cout << "\nPerson #" << (j+1);
        p.diskIn(j);    // read person from disk
        p.showData();    // display person
    }
}

```

C++ ve NESNEYE DAYALI PROGRAMLAMA

283

Overloading the << and >> Operators

In this session I'll show how to overload the extraction and insertion operators. This is a powerful feature of C++. It lets you treat I/O for user-defined data types in the same way as for basic types such as int and double. For example, if you have an object of class ComplexT called c1, you can display it with the statement

```
cout << c1;
```

just as if it were a basic data type.

You can overload the extraction and insertion operators so they work with the display and keyboard (cout and cin). With a little more care, you can also overload them so they work with disk files as well.

```

#include<iostream.h>
class ComplexT{
    float re,im;
    friend istream& operator >>(istream&, ComplexT&);
    friend ostream& operator <<(ostream&, const ComplexT&);
public:
    ComplexT(float re_in=0,float im_in=0){re=re_in;im=im_in;}
    ComplexT operator+(const ComplexT&);
};

```

C++ ve NESNEYE DAYALI PROGRAMLAMA

284

```

istream& operator >>(istream& stream, ComplexT& z)    // Overloading >>
{
    cout << "Enter real part:";
    stream >> z.re;
    cout << "Enter imager part:";
    stream >> z.im;
    return stream;
};
ostream& operator <<(ostream& stream, const ComplexT& z) // Overloading <<
{
    stream << "( " << z.re << " , " << z.im << " ) \n";
    return stream;
};
ComplexT ComplexT::operator+(const ComplexT& z)    // Operator +
{
    return ComplexT(re+z.re , im+z.im);
}
void main()
{
    ComplexT z1,z2,z3;
    cin>>z1;
    cin>>z2;
    z3=z1+z2;
    cout << " Result=" << z3;
}

```

See Example: inout.cpp

C++ ve NESNEYE DAYALI PROGRAMLAMA

285

Overloading for Files

The next example shows how the << and >> operators can be overloaded so they work with both file I/O and cout and cin.

```

#include<fstream.h>
class ComplexT{
    float re,im;
    friend istream& operator >>(istream&, ComplexT&);
    friend ostream& operator <<(ostream&, const ComplexT&);
public:
    ComplexT(float re_in=0,float im_in=0){re=re_in;im=im_in;}
};
istream& operator >>(istream& stream, ComplexT& z)
{
    char dummy;
    stream >> dummy >> z.re;
    stream >> dummy >> z.im >> dummy;
    return stream;
};
ostream& operator <<(ostream& stream, const ComplexT& z){
    stream << "( " << z.re << " , " << z.im << " ) \n";
    return stream;
};

```

C++ ve NESNEYE DAYALI PROGRAMLAMA

286

```

void main()
{
    char ch;
    ComplexT z1;
    ofstream ofile;           // create and open
    ofile.open("complex.dat"); // output stream
    do{
        cout << "\nEnter Complex Number:(re,im)";
        cin >> z1;           // get complex number from user
        ofile << z1;          // write it to output str
        cout << "Do another (y/n)? ";
        cin >> ch;
    }while(ch != 'n');
    ofile.close();           // close output stream
    ifstream ifile;          // create and open
    ifile.open("complex.dat"); // input stream
    cout << "\nContents of disk file is:";
    while(!ifile.eof())
    {
        ifile >> z1;         // read complex number from stream
        if(ifile)
            cout << "\nComplex Number = " << z1; // display complex number
    }
}

```

See Example: fileio.cpp

C++ ve NESNEYE DAYALI PROGRAMLAMA

287

Overloading for Binary I/O

So far, you've seen examples of overloading operator<<() and operator>>() for formatted I/O. They also can be overloaded to perform binary I/O. This may be a more efficient way to store information, especially if your object contains much numerical data.

```

#include <fstream.h> // for file streams
class person // class of persons
{
protected:
    char name[40]; // person's name
    int age; // person's age
public:
    void getData() // get data from keyboard
    {
        cout << "\n Enter name: "; cin.getline(name, 40);
        cout << " Enter age: "; cin >> age;
    }
    void putData() // display data on screen
    {
        cout << "\n Name = " << name; cout << "\n Age = " << age;
    }
    friend istream& operator >> (istream& s, person& d);
    friend ostream& operator << (ostream& s, person& d);
}

```

C++ ve NESNEYE DAYALI PROGRAMLAMA

288


```

void persin(istream& s) // read file into ourself
{
    s.read( (char*)this, sizeof(*this) );
}
void persout(ostream& s) // write our data to file
{
    s.write( (char*)this, sizeof(*this) );
}
}; // end of class definiton

istream& operator >> (istream& s, person& d) // get data from disk
{
    d.persin(s);
    return s;
}

ostream& operator << (ostream& s, person& d) // write data to disk
{
    d.persout(s);
    return s;
}

```

C++ ve NESNEYE DAYALI PROGRAMLAMA

289

```

void main()
{
    // create 4 persons
    person pers1, pers2, pers3, pers4;
    cout << "\nPerson 1";
    pers1.getData(); // get data for pers1
    cout << "\nPerson 2";
    pers2.getData(); // get data for pers2
    outfile("PERSON.DAT", ios::binary); // create output stream
    ofstream
    outfile << pers1 << pers2; // write to file
    outfile.close();
    ifstream infile("PERSON.DAT", ios::binary); // create input stream
    infile >> pers3 >> pers4; // read from file into
    cout << "\nPerson 3"; // pers3 and pers4
    pers3.putData(); // display new objects
    cout << "\nPerson 4";
    pers4.putData();
}

```

C++ ve NESNEYE DAYALI PROGRAMLAMA

290

Exceptions

Program Errors

- Kinds of errors with programs
 - Poor logic - bad algorithm
 - Improper syntax - bad implementation
 - Exceptions - Unusual, but predictable problems
- The earlier you find an error, the less it costs to fix it
- Modern compilers find errors early

Paradigm Shift from C

- In C, the default response to an error is to continue, possibly generating a message
- In C++, the default response to an error is to terminate the program
- C++ programs are more “brittle”, and you have to strive to get them to work correctly
- Can catch all errors and continue as C does

assert()

- a macro (processed by the precompiler)
 - Returns TRUE if its parameter is TRUE
 - Takes an action if it is FALSE
 - abort the program
 - throw an exception
- If DEBUG is not defined, asserts are collapsed so that they generate no code

assert() (cont'd)

- When writing your program, if you know something is true, you can use an assert
- If you have a function which is passed a pointer, you can do
 - `assert(pTruck);`
 - if `pTruck` is 0, the assertion will fail
- Use of `assert` can provide the code reader with insight to your train of thought

assert() (cont'd)

- `Assert` is only used to find programming errors
- Runtime errors are handled with exceptions
 - `DEBUG` false => no code generated for `assert`
 - `Animal *pCat = new Cat;`
 - `assert(pCat);` // bad use of `assert`
 - `pCat->memberFunction();`

assert() (cont'd)

- assert() can be helpful
- Don't overuse it
- Don't forget that it "instruments" your code
 - invalidates unit test when you turn DEBUG off
- Use the debugger to find errors

Exceptions

- You can fix poor logic (code reviews, debugger)
- You can fix improper syntax (asserts, debugger)
- You have to live with exceptions
 - Run out of resources (memory, disk space)
 - User enters bad data
 - Floppy disk goes bad

Why are Exceptions Needed?

- The types of problems which cause exceptions (running out of resources, bad disk drive) are found at a low level (say in a device driver)
- The low level code implementer does not know what your application wants to do when the problem occurs, so s/he “throws” the problem “up” to you

How To Deal With Exceptions

- Crash the program
- Display a message and exit
- Display a message and allow the user to continue
- Correct the problem and continue without disturbing the user

Murphy's Law: "Never test for a system error you don't know how to handle."

What is a C++ Exception?

- An object
 - passed from the area where the problem occurs
 - passed to the area where the problem is handled
- The type of object determines which **exception handler** will be used

Syntax

```
try {  
    // a block of code which might generate an exception  
}  
catch(xNoDisk) {  
    // the exception handler(tell the user to  
    // insert a disk)  
}  
catch(xNoMemory) {  
    // another exception handler for this "try block"  
}
```

The Exception Class

- Defined like any other class:
 - `class Set {`
 - `private:`
 - `int *pData;`
 - `public:`
 - `...`
 - `class xBadIndex {}; // just like any other class`
 - `};`

Throwing An Exception

- In your code where you reach an error node:
 - `if (memberIndex < 0)`
 - `throw xBadIndex();`
- Exception processing now looks for a *catch block* which can handle your thrown object
- If there is no corresponding catch block in the immediate context, the ***call stack*** is examined

The Call Stack

- As your program executes, and functions are called, the return address for each function is stored on a push down stack
- At runtime, the program uses the stack to return to the calling function
- Exception handling uses it to find a catch block

Passing The Exception

- The exception is passed up the call stack until an appropriate catch block is found
- As the exception is passed up, the destructors for objects on the data stack are called
- There is no going back once the exception is ***raised***

Handling The Exception

- Once an appropriate catch block is found, the code in the catch block is executed
- Control is then given to the statement after the group of catch blocks
- Only the active handler most recently encountered in the thread of control will be invoked

Handling The Exception (cont'd)

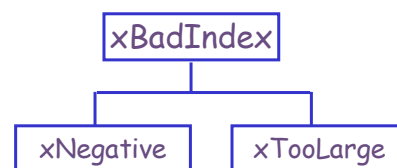
- `catch (Set::xBadIndex) {`
- `// display an error message`
- `}`
- `catch (Set::xBadData) {`
- `// handle this other exception`
- `}`
- `//control is given back here`
- If no appropriate catch block is found, and the stack is at main(), the program exits

Default catch Specifications

- Similar to the switch statement
 - `catch (Set::xBadIndex)`
 - `{ // display an error message }`
 - `catch (Set::xBadData)`
 - `{ // handle this other exception }`
 - `catch (...)`
 - `{ // handle any other exception }`

Exception Hierarchies

- Exception classes are just like every other class; you can derive classes from them
- So one try/catch block might catch all bad indices, and another might catch only negative bad indices



Exception Hierarchies (cont'd)

```
class Set {
private:
    int *pData;
public:
    class xBadIndex {};
    class xNegative : public xBadIndex {};
    class xTooLarge: public xBadIndex {};
};

// throwing xNegative will be
// caught by xBadIndex, too
```

Data in Exceptions

- Since Exceptions are just like other classes, they can have data and member functions
- You can pass data along with the exception object
- An example is to pass an error subtype
- for xBadIndex, you could throw the type of bad index

Data in Exceptions (Continued)

```
// Add member data,ctor,dctor,accessor method
class xBadIndex {
private:
    int badIndex;
public:
    xBadIndex(int iType):badIndex(iType) {}
    int GetBadIndex () { return badIndex; }
    ~xBadIndex() {}
};
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

313

Passing Data In Exceptions

```
// the place in the code where the index is used
if (index < 0)
    throw xBadIndex(index);
if (index > MAX)
    throw xBadIndex(index);
// index is ok
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

314

Getting Data From Exceptions

```
catch (Set::xBadIndex theException)
{
    int badIndex = theException.GetBadIndex();
    if (badIndex < 0 )
        cout << "Set Index " << badIndex << " less than 0";
    else
        cout << "Set Index " << badIndex << " too large";
    cout << endl;
}
```

Caution

- When you write an exception handler, stay aware of the problem that caused it
- Example: if the exception handler is for an out of memory condition, you shouldn't have statements in your exception object constructor which allocate memory

Exceptions With Templates

- You can create a single exception for all instances of a template
 - declare the exception outside of the template
- You can create an exception for each instance of the template
 - declare the exception inside the template

Single Template Exception

```
class xSingleException {};  
  
template <class T>  
class Set {  
private:  
    T *pType;  
public:  
    Set();  
    T& operator[] (int index) const;  
};
```

Each Template Exception

```
template <class T>
class Set {
private:
    T *pType;
public:
    class xEachException {};
    T& operator[] (int index) const;
};
// throw xEachException();
```

C++ ve NESNEYE DAYALI PROGRAMLAMA

319

Catching Template Exceptions

- Single Exception (declared outside the template class)
 - catch (xSingleException)
- Each Exception (declared inside the template class)
 - catch (Set<int>::xEachException)

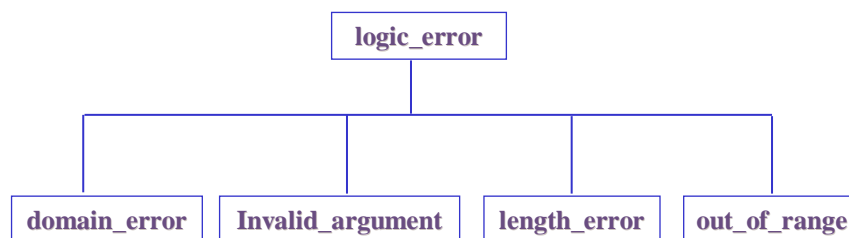
C++ ve NESNEYE DAYALI PROGRAMLAMA

320

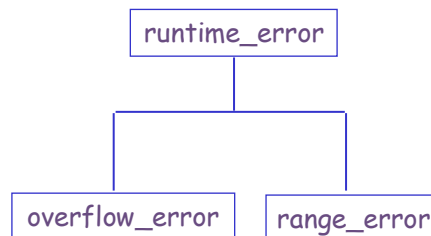
Standard Exceptions

- The C++ standard includes some predefined exceptions, in `<stdexcept.h>`
- The base class is `exception`
- Subclass `logic_error` is for errors which could have been avoided by writing the program differently
- Subclass `runtime_error` is for other errors

Logic Error Hierarchy



Runtime Error Hierarchy



The idea is to use one of the specific classes (e.g. `range_error`) to generate an exception

Data For Standard Exceptions

```
// standard exceptions allow you to specify
// string information
throw overflow_error("Doing float division in function div");

// the exceptions all have the form:
class overflow_error : public runtime_error
{
public:
    overflow_error(const string& what_arg)
        : runtime_error(what_arg) {};
```

Catching Standard Exceptions

```
catch (overflow_error)
{
    cout << "Overflow error" << endl;
}

catch (exception& e)
{
    cout << typeid(e).name() << ": " << e.what() << endl;
}
```

More Standard Exception Data

- `catch (exception& e)`
- Catches all classes derived from *exception*
- If the argument was of type *exception*, it would be converted from the derived class to the exception class
- The handler gets a *reference to exception* as an argument, so it can look at the object

typeid

- **typeid** is an operator which allows you to access the type of an object at runtime
- This is useful for pointers to derived classes
- **typeid** overloads ==, !=, and defines a member function *name*
- ```
if (typeid(*carType) == typeid(Ford))
```
- ```
    cout << "This is a Ford" << endl;
```

typeid().name

```
cout << typeid(*carType).name() << endl;
// If we had said:
// carType = new Ford();
// The output would be:
// Ford
```

- **So:**

```
    cout << typeid(e).name()
```


returns the name of the exception

e.what()

- The class *exception* has a member function ***what***
- `virtual char* what();`
- This is inherited by the derived classes
- `what()` returns the character string specified in the throw statement for the exception

`throw`

```
overflow_error("Doing float division in function div");  
cout << typeid(e).name() << ": " << e.what() << endl;
```

Deriving New *exception* Classes

```
class xBadIndex : public runtime_error {  
public  
    xBadIndex(const char *what_arg = "Bad Index")  
        : runtime_error(what_arg) {}  
};  
// we inherit the virtual function what  
// default supplementary information character string
```

```

template <class T>
class Array{
    private:
        T *data ;
        int Size ;
    public:
        Array(void);
        Array(int);
        class eNegativeIndex{};
        class eOutOfBounds{};
        class eEmptyArray{};
        T& operator[](int) ;
};

```

C++ ve NESNEYE DAYALI PROGRAMLAMA

331

```

template <class T>
Array<T>::Array(void){
    data = NULL ;
    Size = 0 ;
}

template <class T>
Array<T>::Array(int size){
    Size = size ;
    data = new T[Size] ;
}

```

C++ ve NESNEYE DAYALI PROGRAMLAMA

332

```

template <class T>
T& Array<T>::operator[](int index){
    if( data == NULL ) throw eEmptyArray() ;
    if(index < 0) throw eNegativeIndex() ;
    if(index >= Size) throw eOutOfBounds() ;
    return data[index] ;
}

```

```

Array<int> a(10) ;
try{
    int b = a[200] ;
}
catch(Array<int>::eEmptyArray){
    cout << "Empty Array" ;
}
catch(Array<int>::eNegativeIndex){
    cout << "Negative Array" ;
}
catch(Array<int>::eOutOfBounds){
    cout << "Out of bounds" ;
}

```